

# **A MICROPROGRAMMED APL IMPLEMENTATION**

**RODNAY ZAKS**





# **A MICROPROGRAMMED APL IMPLEMENTATION**

**by Rodney Zaks**

**SYBEX**

2020 Milvia  
Berkeley, Ca. 94704

**SYBEX-EUROPE**

313 rue Lecourbe  
75015 Paris, France

Every effort has been made to provide complete and accurate programs. However, no liability shall be incurred for their correct execution. This work is presented as an educational aid.

Copyright © 1978 SYBEX, Inc. World rights reserved. No part of this publication may be stored in a retrieval system, copied, transmitted, or reproduced in anyway, including, but not limited to, photocopy, photography, magnetic or other recording, without the prior written permission of the publisher.

Library of Congress Card Number: 78-58355

ISBN Number: 89588-005-9

Printed in the United States of America

Printing: 10 9 8 7 6 5 4 3 2

# PREFACE

This book presents the complete theory and design of an interpreter for APL. One of its essential characteristics is to be implemented within 2K ROM locations. This was made possible by an in-depth analysis of the intrinsic syntactic characteristics of the language. This analysis resulted in such a terse interpreter. Additional RAM memory is needed to store least frequently used APL operators, which are implemented as APL functions.

In the first part, the complete theory is presented, along with ALGOL models of the parser. The actual details of the implementation are then described, including the listing for a DSC Meta 4: design of the parser, syntactic analysis, data and program representation, function mechanics, array management, memory management, APL operators, including floating-point, n-dimensional arrays, and descriptors. The system grants every user 64K virtual memory, regardless of actual RAM available.

The program can be transcoded into any "standard" instruction-set, such as an 8080, 6800, or other, with reasonable ease.

In addition, the reader should gain a thorough understanding of the conceptual and practical problems associated with the parsing, interpretation and execution of a powerful lambda-language such as APL.



# CONTENTS

Illustrations	vii
Chapter I - Introduction	2
Chapter II - Parsing APL as a "Nested Primitive Language"	
1. Introduction	6
2. Definitions	7
3. Primitive Languages	8
4. Nested Primitive Languages	10
5. Parsing a Nested Primitive Language	11
6. APL as a Nested Primitive Language	17
7. Semantic Auto-Specification	39
Chapter III - Translation From External to Internal Representation	
1. Internal APL Representation	42
2. Phantoms	45
3. Syntax Descriptors	47
4. Static Structures	59
5. The Translator	62
Chapter IV - Dynamic APL Structures - The Interpreter	
1. Introduction	70
2. A New Functional Structure	70
3. Dynamic Memory Structures	75
4. Function Reference Zone	81
5. Dynamic Array Management	85
6. The Interpreter	98

## Chapter V - The Operations

1. APL Operators	118
2. A Front End Processor for Scalar Operators	118
3. Other Operators	120
4. Indexing as a Dyadic Operator	122

## Chapter VI - Perspective

1. General	144
2. The Evolution of Microprogramming	145
3. A System Architecture	157
4. A Language Machine	159
5. Other Processors	160
6. Performance	165
7. Conclusions	172

## Appendices

A. The Syntax of a Primitive Programming Language in BNF	177
B. Simplified Syntax for Meta-APL in BNF	180
C. A Phantom Processor	185
D. The Translator	188
E. Listing of the Translator in Snobol	192
F. Representation, Encoding, and Implementation of APL Operators	209
G. Listings of the APL Operators (Ted Stohr)	214
H. Table of Floating Point Integers (octal, binary, decimal)	227
I. Control Instructions for the Meta-4 Computer	229
J. The Map	247
K. Map Control	251
L. Input-Output Registers	254



M. Listings of the Microprogrammed Interpreter (prototype)	256
N. Complete Timing for Array Addition	320
O. Timing of a Recursive Function	326
References	336
Index	342

# ILLUSTRATIONS

## FIGURE

2.1	A Minimal FSM for NPL Parsing	14
2.2	The Transition Table for the NPL Automaton	15
2.3	Interstate Traffic for the APL Parser	21
2.4	Cycles in the APL Parser	22
2.5	Register Management during Parsing	27
2.6	Operation of the APL Parser	37
3.1	A Phantom Token	46
3.2	Program Strings Descriptors	48
3.3	Scalar	49
3.4	Literal Array	51
3.5	Phantom	52
3.6	Operator	53
3.7	Noop	55
3.8	Input-Output Device	56
3.9	Command	57
3.10	Right Brace	58
3.11	Program Strings: Static Structure for Functions	61
3.12	The Factorial Function: External APL	66
3.13	Factorial: The Internal Strings and Static Structures	68
4.1	APL's Inter-Function Communication	72
4.2	Meta-APL's Inter-Function Communication	73
4.3	Stack Descriptors	76
4.4	Array Block	77

## FIGURE

4.5	Virtual Memory Management	80
4.6	Phantom Materialization Segment (PMS)	83
4.7	Local Function Reference Zone (LFRZ)	84
4.8	The Floating Bit	93
4.9	Storage Allocation	95
4.10	Storage Deallocation	96
4.11	Factorial: Translator Input	108
4.12	Factorial: Translator Output	109
4.13	The First Call is Encountered	110
4.14	The Second Call is Generated	111
4.15	Third Call	112
4.16	Fourth and Final Call	113
4.17	The Returns	114
4.18	Factorial 7	115
4.19	Factorial 1	116
5.1	Data Structures for APL	125
5.2	Row Major Order Retrieval	129
5.3	The Odometer	131
5.4	Final Odometer	131
5.5	The Two-Slot Odometer	132
5.6	The Odometer Blocks	142
6.1	Conceptual Model of a Microprocessor	148
6.2	Timing of Micro-operations	151
6.3	Physical Connections for a Logical IO or Memory Slot	152
6.4	Timings for Addition of Two 5-Element Arrays	166
6.5	Timings in Array-Array Operations	167
6.6	Size of the Routines (# of instructions)	171



# **chapter 1**

## **i n t r o d u c t i o n**

## INTRODUCTION

APL ("A Programming Language") was originally developed by Iverson [ ] as a programming language-like notation designed for the terse specification of numerical and logical algorithms. It was subsequently implemented by Iverson, Falkoff and Breed as a full programming language for IBM Corporation and released as APL/360 in the Summer of 1968. This system, specifically oriented towards algorithmic specification and interactive use led rapidly to the development or release of other implementations: IBM 1130 [7], XDS Sigma 7 [4], Burroughs B5500, DEC PDP10. Furthermore, several companies have entered the time-sharing field with an exclusive APL service, demonstrating the increasing dissemination of the language.

APL is a high-level language characterized by a minimal syntactic complexity (no operator precedence, no data structure declarations), where the traditional bookkeeping overhead associated with the execution of complex operators has been completely internalized. It provides a large number of semantic operators, represented by special symbols, which can be implicitly extended to manipulate arrays of any dimensionality. The language has proven to be an excellent medium for terse specification of complex numerical algorithms.

Until now, APL service has only been available on high-cost computer systems. The dissemination of the language and its availability to small user groups has been limited by the problem of devising a cost-effective hardware-software system. However, the burgeoning development of low-cost "mini"-computers now makes it economically feasible to dedicate one or more processors

to high-level language execution.

This approach permits a technological efficiency not possible in "general-purpose" computing installations. Microprogramming the interpreter permits execution speeds to be achieved which rival the largest conventional implementations. This also does away completely with the overhead inefficiency engendered by the multiple software layers of conventional systems. Furthermore, a dedicated microprocessor offers the possibility of efficient language execution on a minimum hardware, since the processor structure can be optimized for the interpreter.

However, the limited size of current control memories imposes limitations on the complexity of the resident microprogrammed interpretive system.

It is the purpose of this dissertation to present solutions to the problems which arise from the space limitation and the structure of APL.

The solutions take the form of extensive rationalization of phases of the interpretive process, which heretofore had remained cumbersome. This rationalization is accomplished by analyzing the various phases from an automata theoretic point of view and applying the resulting concepts to program construction.

It was found that the area in which specially significant advances had been achieved was in the design of the parser. The minimization of the parsing phase resulted from the recognition of certain properties of the APL language. These properties emerged from the formalization proposed in Chapter II on "Nested Primitive Languages." A second area of significant gain was in the virtual

memory management algorithms. The global syntactic structure of APL induces a "synchronicity" of allocation which is recognized in the implementation. The difficulties arising from the absence of declarations in APL and the dynamic nesting of structures necessitated for parsing have up to this point been approached in an essentially ad hoc fashion. In this implementation, the space limitation forced a complete rationalization of the management algorithms. This topic is presented in Chapter IV - Dynamic APL Structures.

This dissertation also presents some other areas in which insights and consequent advances have been possible: translation from external to internal representation (Chapter III) and design of operators routines (Chapter V). In the last chapter, the system performance is evaluated. The efficiency of this implementation appears to rival large APL systems. Its extreme terseness makes it suitable for implementation even on very small processors. It is hoped that this research will be of value both to language and to systems designers.



## **CHAPTER        2**

### **PARSING APL AS A "NESTED**

### **PRIMITIVE        LANGUAGE"**

1.    Introduction
2.    Definitions
3.    Primitive Languages
4.    Nested Primitive Languages
5.    Parsing a Nested Primitive Language
6.    APL as a Nested Primitive Language
7.    Semantic Autospecification

## 1. Introduction

Syntactic analysis and semantic execution present two central problems in the construction of a high-level language interpreter. Efforts directed at the resolution of these problems in the general case have indicated the need for a simple language specification if efficient implementation is to be achieved.

The special parsing properties associated with a distinguished class of programming languages called primitive programming languages are presented here. A primitive grammar is defined using a set of four productions, the corresponding primitive language being a context-free operator language.

A precedence level is added to this primitive grammar by the introduction of parentheses and permits the construction of a nested primitive language (NPL) or APL-like language. The analysis of the parsing properties for this class of languages leads to the design of a minimal finite-state recognizer. A quantitative measure of parsing complexity for this class of languages is obtained from the automaton and a comparison is established with precedence languages. Parsing complexity for nested primitive languages is shown to be close to a theoretical minimum.

Finally nested primitive languages provide a theoretical and consistent model for APL-like languages and APL is reduced through syntactic partitioning to a nested primitive language. This results in the construction of a terse three-state parser for APL.

## 2. Definitions

The essential concepts necessary for the definition of a phase-structure grammar and of a programming language are introduced here. This permits the presentation in the following sections of a formal definition for the Primitive Languages.

Let  $N$  be the alphabet of non-terminal symbols,  $T$  the alphabet of terminal symbols,  $R$  the set of productions,  $\emptyset$  the empty set, and  $S$  the start symbol.

A *grammar*  $G$  is an ordered quadruple  $(N, T, R, S)$  with  $S \in N$ ,  $N \cap T = \emptyset$ .  $S$  never occurs on the right of a production of  $R$ . A *language*  $L(G)$  is the set of all strings in  $T$  which are generated or recognized by  $G$ . A *programming language*  $PL(G, \xi)$  is a language  $L(G)$  with  $G = (N, T, R, S)$  equipped with a set  $\xi$  of semantic evaluation rules. A *production* of  $R$ ,  $U \rightarrow u$ , is an ordered pair with  $U \in N$ ,  $u \in T$ . For any set  $Z$ ,  $Z^*$  denotes the set of all strings of finite length over  $Z$ .

The set of productions ( $R$ ) is finite;  $U$  is called the left part and  $u$  the right part of the production.

We write  $U_1 \Rightarrow U_2$  if  $\exists \alpha, \beta, \gamma \in (N \cup T)^*$  and  $A \in N$  such that

$$U_1 = \alpha A \beta$$

$$U_2 = \alpha \gamma \beta$$

and  $(A \rightarrow \gamma) \in R$ .

$U_2$  is called the *direct derivative* of  $U_1$ .

$U_1 \xRightarrow{*} U_2$  is the *reflexive-transitive closure* of the relation  $U_1 \Rightarrow U_2$ . A string  $x \in T^*$  such that  $S \xRightarrow{*} x$  is a *sentential form*. The *canonical reduction sequence* for  $x$  is the sequence  $\{\alpha_n, \alpha_{n-1}, \dots, \alpha_1\}$  such that  $\alpha_n = x$ ,  $\alpha_1 = S$ ,  $\alpha_i \Rightarrow \alpha_{i+1}$  ( $i = 1, \dots, n-1$ ) with  $\beta \in T^*$  instead of  $(N \cup T)^*$ .

A language is defined as a set of sentences, i.e., sentential forms consisting only of terminal symbols:

$$L(G) = \{u \mid S \xRightarrow{*} u \wedge u \in T^*\}.$$

Given a sentence, the operation consisting of finding a derivation for it and constructing the corresponding syntax tree is called *parsing*.

Further restricting the elements of  $T$  to appear only on the right of a production, the grammar  $(N, T, R, S)$  is a *phrase structure grammar*, and if:

$$\nexists r \in R \mid r \rightarrow (U \rightarrow u_1 U_1 U_2) \wedge (U, U_1, U_2 \in N) \wedge (u, v \in T),$$

the grammar is an *operator grammar*.

It is easily shown that in an operator grammar, no sentential form contains two adjacent non-terminal symbols.

A well-known distinguished subclass of the operator grammars in the class of *precedence grammars* [Floyd, 1963; Wirth and Weber, 1966]. We have presented the terminology necessary for the formal specification of more complex constructs and will define a new class of operator languages, the Primitive Languages.

### 3. Primitive Languages

Primitive languages are formally defined in this section and will be used as the basis for the construction of the Nested Primitive Languages to be introduced in section 4.

Let "o" denote any element within the set of "operand-types"  $OC$ :

$$OC = \{o_1, o_2, \dots, o_n\}$$

Let "f" denote any element within the set of "function-types"  $FC$ :

$$FC = \{f_1, f_2, \dots, f_m\}$$

A primitive grammar is now defined as the quadruple  $(N, T, R, S)$  with a terminal alphabet  $T$  such that:

$$T = \emptyset \cup \mathcal{H} \text{ and } \emptyset \cap \mathcal{H} = \emptyset,$$

$$N = (S, E),$$

and the following set of productions:

1.  $S \rightarrow E$
2.  $E \rightarrow o$
3.  $E \rightarrow fE$
4.  $E \rightarrow o f E$

(or, in Kleene notation:

$$S \rightarrow \{f^* o f\}^* f^* o).$$

Clearly, the set of *primitive languages* (PL) recognized by the above grammar is a subclass of the (context-free) operator languages.

A *primitive programming language* (PPL) is defined as a primitive language with a strictly monodirectional evaluation rule. Without loss of generality, it can be assumed that evaluation proceeds from right to left. Note that the absence of precedence is reflected by the one-level depth of expression definition before recursion or:  $\text{size}(N) = 2$ .

Equivalently, a PPL's syntax can be defined in BNF, where  $::=$  is used for  $\rightarrow$ , and  $|$  is used for the specification of multiple right parts corresponding to the same left part. A complete example is presented in Appendix A

(p.177). The segment corresponding to the PL specification is:

$\langle \text{STATEMENT} \rangle ::= \langle \text{EXPRESSION} \rangle$

1.1  $\langle \text{EXPRESSION} \rangle ::= \langle \text{OPERAND-TYPE} \rangle$

$|\{ \langle \text{EXPRESSION} \rangle \} \langle \text{FUNCTION-TYPE} \rangle \langle \text{EXPRESSION} \rangle$

where the meta-symbols  $\{ \}$  stand for 0 or 1 time. We now construct the class of nested primitive languages.

#### 4. Nested Primitive Languages

A nested primitive programming language (NPL) or APL-like language is defined as a PPL with a fifth production:

5.  $E \rightarrow (E)$

and a second semantic rule<sup>1</sup> evaluation of parenthesized expressions takes precedence over simple expressions  $E$ . This naturally applies only for a production of type 4 and introduces a level of precedence within the language. The class of operators however remains uniform with respect to precedence, parentheses being special terminals.

The corresponding BNF characterization is modified as follows:

1.1'  $\langle \text{EXPRESSION} \rangle ::= \langle \text{OPERAND-TYPE} \rangle$

$|\{ \langle \text{EXPRESSION} \rangle \} \langle \text{FUNCTION-TYPE} \rangle \langle \text{EXPRESSION} \rangle$

$|(\langle \text{EXPRESSION} \rangle)$  .

On an operator grammar, evaluation is effected through the transitive application of semantic operator routines to the operands in a specified order. In most programming languages, this order of evaluation is essentially determined by parentheses and operator-precedence relationships. Reduction should proceed in the order of evaluation and its complexity is thus directly related to the precedence specification.

---

<sup>1</sup>The first one being the monodirectional evaluation rule.

In the next section are examined the parsing properties associated with primitive languages, where evaluation is completely specified by the two rules introduced above.

### 5. Parsing a Nested Primitive Language

Bounded context parsers have been proposed by [Floyd, 1963] and [Irons, 1964] and consist of an algorithm consulting a table. The table however has to be constructed and several algorithms for doing it in the case of an arbitrary context-free grammar, when such a table exists, have been designed [Loeckx, 1970]. [Knuth, 1965], [Korenjak, 1969] propose methods for constructing LR(k) and in particular LR(1) processors using a partitioning scheme. These methods have the value of generality with its drawbacks: they do not lead to efficient algorithms. Focussing here on a restricted class of languages, the nested primitive languages, it is shown that such a grammar is RL(1), i.e., that it can be parsed during a single deterministic scan from right (R) to left (L) without context analysis (look-ahead) of more than one symbol.

A minimal finite state automaton for NPL parsing is developed and a quantitative measure of the complexity of the parser is obtained by enumerating its transition functions. This result is compared to the ones obtained for ALGOL-like languages.

The parsing strategy for our basic language, the PPL, is first considered.

The *PPL* parser scans the input string from right to left and attempts to recognize expressions. Only two types of terminal symbols, namely o's and f's may appear within the string. The simplicity of this parser derives from the fact that it identifies only two syntactic types: "o" or

"f" and differentiates between productions 3 and 4 through a one-level context analysis. The head, therefore, always assumes one of two states: "o"-found or "f"-found and can be used simultaneously as a top down predictor: in "o" state, "f" is expected, else end of string or illegal syntax. In "f" state, an "f" identifies a previous production 3, while an "o" identifies an actual production 4. The parsing procedure for a PPL is the following:

```
procedure parse(statement);
  begin
    token:= getnext(statement)
    if type(token)  $\neq$  0 then goto syntaxerror;
    token:= getnext(statement);
  next: if type(token)  $\neq$  1 then if token = nil then goto exit else
        goto syntaxerror;
    token:= getnext(statement);
    if type(token) = 0 then goto binaryexpression else goto
        unaryexpression
  end;
```

where procedure getnext returns the next token (syntactic element) of the string, i.e., the token to the left, and the value nil if none is found.

Procedure type(token) returns the value 0 for an operand-type, 1 for a function-type.

Note that the parser differentiates between the three types of expressions (productions 2,3,4) i.e., niladic, monadic, dyadic and branches to the appropriate entry point where the semantic routine corresponding to the operator will take over, apply the operator, leave a result (operand-type) and re-enter the parsing routine at "next".



Let "fo" denote the catenation of an "f" and an "o".

*Parsing an NPL* introduces explicit expression-level nesting and requires a stack structure for temporary storage of "fo"'s whose evaluation must be deferred. Two more syntactic types need be distinguished: a right delimiter: ")" and a left delimiter "(". A right delimiter causes stacking of the pending "fo" if any and re-entry to the parser. A left delimiter causes execution, followed by unstacking of an "fo" if any from the stack.

•states are: 0 :start → "o" wanted  
 1 : "o" found → "f" or "(" wanted  
 2 : "f" found → look-ahead needed

•type (token) returns: 0 for "o"  
 1 for "f"  
 2 for "("  
 3 for ")"

•the procedure now is:

procedure NPLparse(statement);

begin

state 0: state:= 0 ;

token:= getnext(statement); subexpression[1]:= token;

if type(token) = 1 then goto syntaxerror

else if type(token) = 2 then goto syntaxerror

else if type(token) = 3 then goto pushdown;

state 1: state:= 1 ;

token:= getnext(statement); subexpression[2]:= token;

if type(token) = 0 then goto syntaxerror

else if type(token) = 3 then goto syntaxerror

else if type(token) = 2 then goto nullaryexpression;

```

state 2:  state:= 2 ;

          token = getnext(statement); subexpression[3]:= token;

          if type(token) = 0 then goto binaryexpression;

          else if type(token) = 1 then goto  unaryexpression;

          else if type(token) = 2 then goto  unaryexpression;

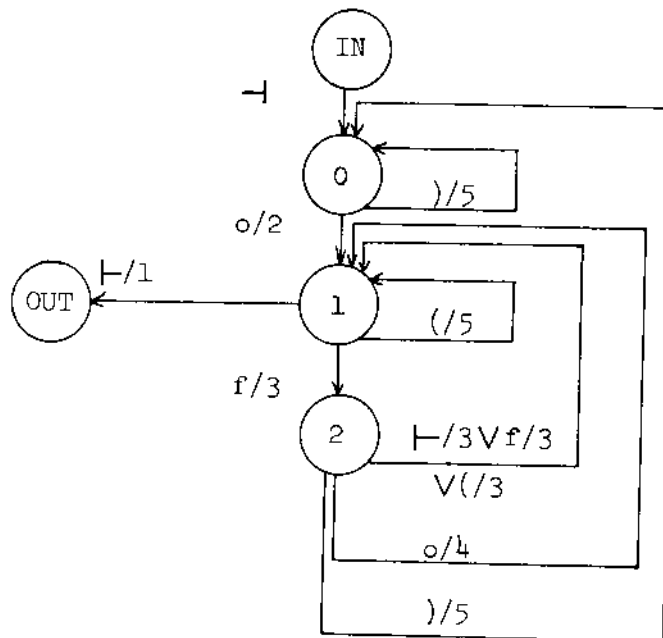
          else goto pushdown;

```

The return state for a unary or binary expression is state 1.

The complete parsing procedure for an NPL has now been presented and a minimal finite-state machine for an NPL's RL(1) processor can be constructed. Without loss of generality, it is assumed  $S \rightarrow \vdash E \vdash$ , the pad symbols being 2 symbols which appear nowhere else:

Figure 2.1 - A Minimal FSM for NPL Parsing



The edges are labeled with the input symbol followed by the number of the production.

A simple and direct measure of the complexity and size of a parsing algorithm is given by the transition functions count on the finite-state automaton, when it can be built.

Figure 2.2 - The Transition Table for the NPL Automaton

input state	production				next state			
	o	f	(	)	o	f	(	)
0	2	6	6	5	1	3	3	0
1	6	3	5	6	3	2	1	3
2	4	3	3	5	1	1	1	0
3	6	6	6	6	3	3	3	3

State 3 and production 6 identify a syntactic error. Counting the transition functions of the machine, without the error logic, i.e., excepting state 3, the input/state transition becomes:

$$\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

where a one denotes an existing transition. The number of transition functions required for NPL recognition is therefore 8.

Comparing this result to the number of state sets necessary for an ALGOL-like language and using Knuth's algorithm, the number of state-sets is of the order of several hundred [Knuth, 1965], [Korenjak, 1969], [Earley, 1969], even after syntax partitioning.

This reveals the practical significance of a Nested Primitive programming language: the extreme terseness of the NPL recognizer permits a compact and efficient interpretation of an APL-like language.

In hierarchical control processors as well as traditional machines, the complexity of interpretation presents one of the most important limiting factors for the design of higher level languages for man-machine communication. This used to be especially true at the hardware level, but the border between hardware and software has been further blurred by the generalized advent of microprogramming, introducing one more level in the control hierarchy. Because of the restricted size of presently available control storage, typically 1,000 to 2,000 instructions, the conciseness of the interpreter may even draw the line as to the feasibility of an implementation.

The properties derived from this study of an NPL language have permitted an extremely terse APL interpreter to reside within such a 2,000-instruction control storage, thus effectively structuring the machine into a direct APL-like language processor.

Finally, as another consequence of this intrinsic simplicity complete syntactic error detection facilities are made available by state 3 of the transition diagram (see also the ALGOL description above for the NPL parser):

- in state 0, a type 1 token is detected as illegal function type  
a type 2 token is an illegal left paren
- in state 1, a type 0 token is an illegal operand  
a type 3 token is an illegal right paren.

The early detection of syntactic errors results from the predictive behavior of the parser and allows total error recovery at the token level: detection occurs before evaluation of the expression level.

These parsing properties of APL-like languages will now be applied to APL.

## 6. APL as a Nested Primitive Language

### 6.1 - General

The parsing automaton developed in the preceding section for Nested Primitive Languages will now be used for the practical realization of an APL parsing algorithm. A syntax partitioning technique has been used for the Nested Primitive Language: productions 1.1 and 1.2 of Appendix A can be viewed as the preprocessed or internal representation of the symbolic programming language. Similarly, a partitioning is used for APL and the partitioned description of the language in BNF appears in Appendix B. The extent and techniques of this preprocessing constitute the topic of Chapter II: Translation from external to internal representation.

For the purposes of this section, the preprocessor or "Translator" is defined as a lexical analyzer for groups 2 and 3 of Appendix B whose output is "Internal APL". Internal APL tokens are essentially of type "o" (operand), "f" (function or operator), or "Φ" (phantom). The syntactic type of phantoms has been incompletely resolved as a result of APL's facility to use identifiers before their definition. Complete details about phantoms are given in the next chapter. This feature requires the use of a special processor, the Phantom Processor, in order to resolve the syntactic ambiguity of "Φ"-tokens into "o"-tokens or "f" tokens. This resolution occurs dynamically at execution time.

62 - APL and NPL

The internal APL strings created by the Translator (see Chapter III, section 5) are composed of nine descriptor types (see Figure 3.2). They correspond to a syntax partitioning effected by the Translator.

These tokens will be parsed by the APL parser. A broad classification of their types is the following:

$\langle \text{OPERAND-TYPE} \rangle ::= \langle \text{scalar} \rangle | \langle \text{array} \rangle | \langle \text{IOD} \rangle | \langle \text{variable} \rangle$

$\langle \text{FUNCTION-TYPE} \rangle ::= \langle \text{operator} \rangle | \langle \text{function} \rangle$

$\langle \text{PHANTOM} \rangle ::= \langle \text{OPERAND TYPE} \rangle | \langle \text{FUNCTION TYPE} \rangle$

$\langle \text{OPERATOR} \rangle ::= \langle \text{numeric operators} \rangle$

APL will be compared to an NPL in order to demonstrate their fundamental family relationship and indicate where APL departs from an NPL.

For clarity, the above classification ignores temporarily NOOP, CO (command) and RB (right brace). It has also limitations which will be indicated below. Recall here the grammar of a nested primitive language:

1.  $S \rightarrow E$
2.  $E \rightarrow o$
3.  $E \rightarrow f E$
4.  $E \rightarrow o f E$
5.  $E \rightarrow (E)$

One property is now apparent: provided that phantoms can be resolved into an "o" (variable) or an "f" (function) type, the APL tokens fundamentally conform to an NPL grammar. In other words, translated APL is essentially an NPL language.

In order to qualify this statement, the restrictions are now considered:

Operators: Special APL operators consisting of double or triple symbols must be treated as a single operator. In practice, this simply means that special operators are to pick up their companion to the left and to the right without changing the state of the parser. This detection is made possible by the special bits of the OP token. Furthermore, some special operators, namely execops, rdelims, and rbrace require special treatment. This topic is addressed in detail in Chapter V and in section 62.

Functions: Only those functions which have arguments appear as an operator type. Those without arguments appear syntactically as an operand type. Furthermore, functions without explicit result (and their possible arguments) must appear alone on a line and cannot be combined in an expression.

In summary: (internal) APL is essentially an NPL language. Special cases are introduced by the special APL facilities described above. These additions to an NPL introduce added parsing complexity. However, the fundamental parsing mechanism is the one developed in section 5. The parsing mechanism for APL is now considered in more detail.

### 63 - Detailed Operation of the APL Parser

Once the resolution of phantoms has been effected, all parsed tokens are of type "o" or "f". The differences between internal APL and a Nested Primitive Language are the following:

- Special "inhomogeneous" operators used in APL require special semantic routines, but no supplementary parsing state. In particular, [---] will be shown to be equivalent to a dyadic operator. ← and › do not require any special syntactic treatment, other than validity checks and are parsed as a dyadic and a monadic operator respectively. ; and ] cause the parser to re-enter state 0 (new expression level) and are deferred like any dyadic operator followed on the left by a right parenthesis. These operators have been grouped as "right delimiters" and "executable operators" for their common semantic actions. They are examined in Section C3 of Chapter IV.

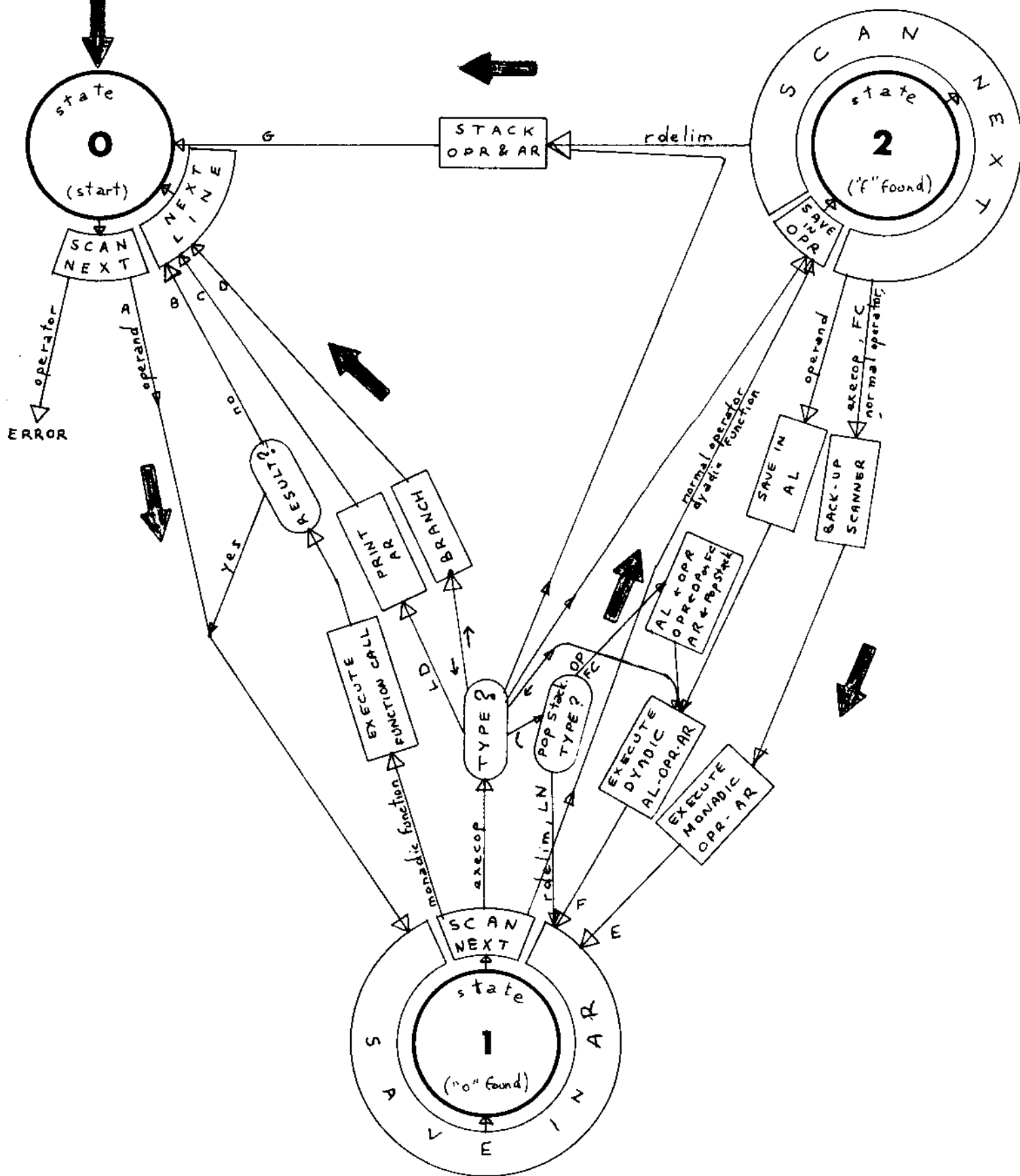
- The existence of functions without explicit result requires a special syntax check since they are neither "o" nor "f". They must constitute the sole entry on the line.

- A line delimiter (LD) is introduced as an extra-type for the internal APL representation in order to delimit the left end of a line. It is an "executable operator" and is syntactically analogous to a monadic operator. Its semantic action results in a value print-out or the absence of it, with the extensions noted above. The Internal APL representation now obeys essentially the syntax and evaluation rules of an NPL and can be parsed by the 3-state NPL parser developed in the preceding section.

A flow diagram of the Parser indicating the traffic between the three states of the Parser is given in figure 2.3. For clarity,



Figure 2.3 - Interstate Traffic in the APL Parser



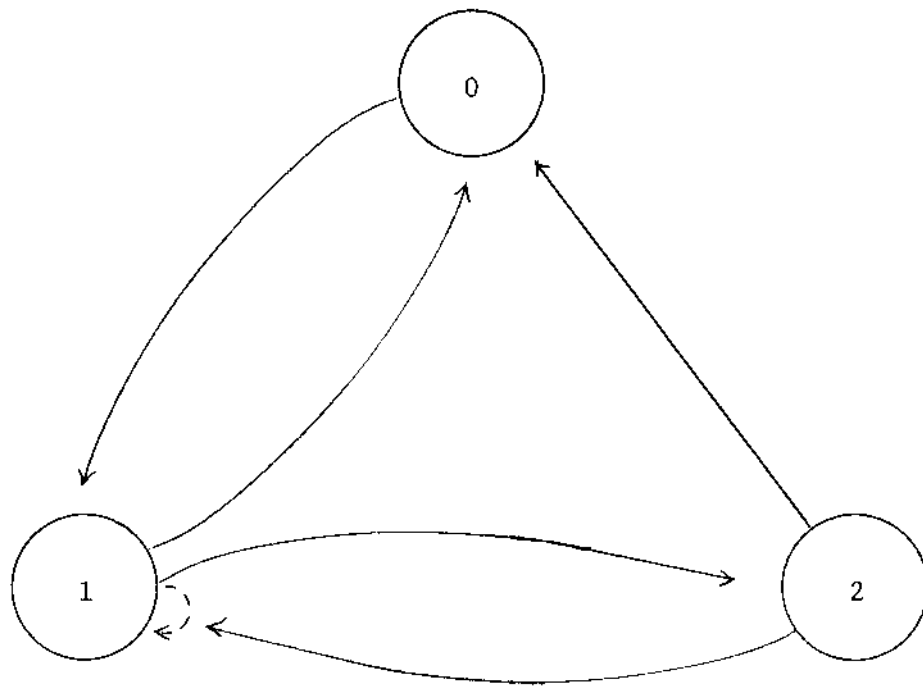


Figure 2.4 - Cycles in the APL Parser

only the essential routines are shown on the diagram. In particular, everytime a Phantom is scanned, the Phantom Processor is called, resulting in a value ("o") or function call ("o" or "f" according to function type).

It must be stressed that at any point in the syntax analysis of an APL sentence, the behavior of the parser can be defined by one of only three states. This can be contrasted to the large number of states used in earlier implementations of APL.

The possible cycles for the parser are apparent on figure 2.3. They are drawn on figure 2.4.

The essential characteristics of the operation of the parser are the following:

- State 0 corresponds to the start of a new expression level. It is entered when starting a line and upon encountering a right delimiter or some execops (; branch).
- State 1 is entered upon detection of an "o" (operand-type) in state 0. This operand is stored in a set of registers labeled AR (right argument). State 1 therefore corresponds to "operand found". It is the return state upon execution of all numeric operators.
- State 2 is entered upon detection of an "f" in state 1. It corresponds to "operator found". The operator is stored in the OPR register. Unless the "f" is a monadic function, the type of the operator-type just scanned cannot be resolved as dyadic or monadic (see Phantoms section), requiring a one-symbol look ahead while in state 2. In the case of a monadic function scanned while in state 1, no look-ahead is needed, and the call may be directly executed.

The diagram of figure 2.3 shows this simplification by a direct arrow from state 1 to state 0 in case of a monadic function call.

In state 2, the next token is examined.

- \* If a right delimiter, the AR and the OPR must be deferred into the Stack (operator-operand pair) and a new expression level is entered (return to state 0).

- \* If another operator or a function, the operator in OPR has been identified as a monadic operator. It is executed, and since it leaves an operand-type result, execution exits in state 1 (arrow E on figure 2.3).

- \* If an operand, it is loaded in AL (left argument) and the dyadic execution routine is called for AL-OPR-AR. A result is left and parsing resumes in state 1 (arrow F).

More details about state 1 can now be given.

In state 1, scanning an "executable operator" results in the following actions:

- \* Branch is handled as a monadic operator which is immediately recognizable. Early detection is necessary in view of the special syntactic check necessary. The next token must be an LD (end of line).

- \* Lbrace ({) is used in conjunction with the special facility proposed in this dissertation for defining parametrized functions (an extension to APL, described in Chapter IV). It matches a previous Rbrace (}).

- \* L bracket necessitates an early special check in order to verify that the top of the stack contains a ; or a ], but is otherwise handled like any other dyadic operator.

\* A line delimiter checks the top of the Stack. If empty (in practice the Stack contains a line number (LN) token created upon line entry), exit to state 0 occurs (next line). Otherwise, the top of the stack is printed along with all other values (if any) which may appear on it, separated by semicolons, until the LN is found.

\* An assignment requires a special syntax check and can be immediately recognized as a dyadic operator. It is therefore executed immediately. The token on the left of the  $\leftarrow$  is verified to be a phantom and to resolve into a value (not a function call) or to be a `]`, in which case the OPR-AR are deferred like any other dyadic operator in state 2.

This short cut is only intended to improve efficiency, but does not detract from the simplicity of the parser.  $\leftarrow$  can be perfectly handled as another dyadic operator.

\* Similarly `;` is immediately identified as a "dyadic" operator and results in deferring the OPR-AR (operator-operand pair) into the stack, bypassing state 2. Parsing resumes in state 0.

This description of the Parser operation should be read again after reading Chapters III, IV and V, which clarify related aspects of operation. As a result, an important criterion in the efficiency of a practical implementation is the ability to implement AL, OPR and AR in fast registers. These registers are used for operator execution: OPR-AR for monadic operators, AL-OPR-AR for dyadic ones. The result is left in AR. OPR and AR are seen to contain the top of the parsing stack when an Rdelim is scanned in state 2. These concepts are illustrated in figure 2.5 below. The parser state is a mere switch between three possible destination registers. This

illustration provides another visualization of the three parsing states necessary and sufficient for APL parsing.

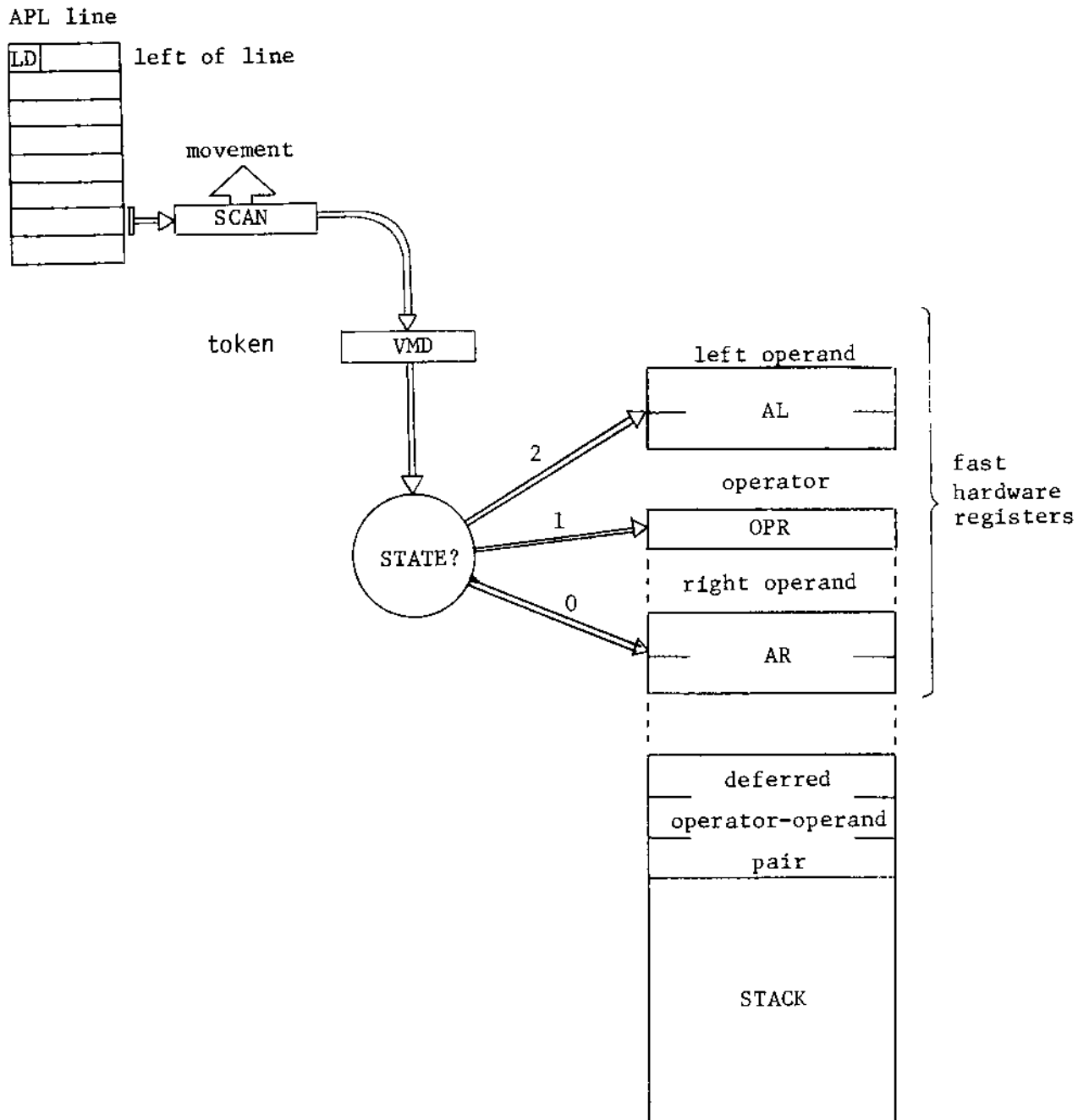


Figure 2.5 - Register Management During Parsing

#### 64 - Algol Description of the Parser

The Algol procedure which parses APL statements is now presented. It is detailed and shows the management of the essential registers (refer to Figure 2.5), the stack management and the syntax checks. Being machine-independent, this parser is useful for the understanding of the microprogrammed implementation and could be used for any other APL implementation.

In order to follow through the APL parser, it is helpful to review the procedure NPLparse in the preceding section, and to observe the parsing mechanism illustrated on Figure 2.5.



### Detailed APL Parser:

```
procedure      APLparse(statement);  
begin  
    switch S := branch, semicln, lbrace, lparen, lbracket,  
LD, assign;  
    switch D := operandtype, operatortype, phiprocessor, special;  
    switch P := operandtype, undefinedvariable, functioncall,  
phiprocessor;  
startline:     stack(LNtoken);  
clear:         state := 0;  
parsenext:     token := getnext(statement);  
decypher:      go to D[type(token)];  
operandtype:   AL := token;  
               if state = 1 then go to syntaxerror  
               else if state = 2 then go to dyadic  
               else begin   state := 1;  
                           AR := token;  
                           go to parsenext end;  
operatortype:  operator := token;  
               if group(operator) = 0 then go to rdelim;  
               if state = 0 then go to syntaxerror  
               else if state = 2 then go to monadic;  
               if group(operator) = 1 then go to execop;  
               else begin   OPR := operator;  
                           state := 2;  
                           go to parsenext end;
```

```

execop:      go to S(operator);
branch:      token := getnext(statement);
              if descriptor(token) # 'LD' then go to syntaxerror;
              reset(linenum);
              go to startline;
semicln:     defer(AR);
              defer(operator);
              go to clear;
lbrace:      go to parsenext;
lparen:      if field(pop) # 'rparen' then go to syntaxerror;
unstack:     token := copytopofstack;
              if group(token) # 0 then go to endtest;
outl:        state := 1;
              go to parsenext;

endtest:     if descriptor(token) = 'LN' then go to outl;
popem:       AL := AR;
              OPR := pop;
              AR := pop;
              go to dyadic;

lbracket:    state := 2;
              token := copytopofstack;
              if field(token) = 'semicln' V 'rbracket' then go to parsenext;
              else begin token := pop; go to popem end;

LD:          endline;

```

```

assign:      token := getnext(statement);
             if type(token) # 2 then go to indexassign;
             state := 1;
             assignit;
             token := getnext(statement);
             if descriptor(token) = 'LD' then go to endline
             else go to decypher;

indexcheck:  defer(OPR,AR);
             push(token);
             state := 0;
             go to parsenext;

rdelim:      if state = 1 then go to syntaxerror
             else if state = 2 then defer(OPR,AR);
             defer(operator);
             go to clear;

phiprocessor: token := lookup(token);
              go to P[phantomtype(token)];

functioncall: if state = 2 then go to fc2
              else if state = 0 then go to paramcheck;
              if argnumber(token) = 0 then go to syntaxerror
              else go to paramcheck;

fc2:         if argnumber(token) = 0 then go to monadic;
              if resultnumber(token) = 0 then go to syntaxerror;

```

```

paramcheck:      verifyparamnumber;
                  if argnumber(token) # 0 then defer(AR)
                  else if argnumber(token) # 2 then go to nuframe
                  else begin   OPR := token;
                                state := 2;
                                go to parsenext end;

nuframe:         setupnewFRZ;
                  linenummer := 0;
                  go to startline;

freturn:         deleteLFRZ;
                  if resultnumber(function) = 0 then go to noresult;
                  AR := result;
                  if argnumber(function) = 0 then go to unstack;
                  state := 1;
                  go to parsenext;

noresult:        token := getnext(statement);
                  if descriptor(token) # 'LD' then go to syntaxerror else
                  go to endline;

special:         if descriptor(token) = 'command' then executecommand
                  else if descriptor(token) = 'NOOP' then go to parsenext
                  else performrbrace;

monadic:         backup(statement);
                  if field(OPR) = 'lbracket' then go to opindex
                  else if field(OPR) = 'dot' then go to dotoper;

```

```

operate:      AR := result(AL,OPR,AR);
              state := 1;
              go to parsenext;

dyadic:      if field(OPR) = 'lbracket' then go to index
              else if descriptor(OPR) = 'FC' then go to dyadicfunctioncall
              else go to operate;

opindex:     OPR := getnext(statement);
              if descriptor(OPR) # 'indexable operator' then
              go to syntaxerror;
              AL := AR;
              token := pop;
              token := copytopofstack;
              if type(token) = 1 then go to operate
              else begin          savedOPR := OPR;
                                   OPR := pop;
                                   AR := pop;
                                   AR := result(OPR,AR);
                                   OPR := savedOPR
                                   go to operate end;

dotoper:     branchto(OPR);

dyadicfunctioncall: AR := AL;
              stack(AR);
              token := OPR;
              go to nuframe;

endline:     token := pop;
              if descriptor(token) = 'semicolon' then go to print

```

```
    else printAR;  
    getnextline;  
    go to startline;  
print:  printAR;  
        go to endline;  
  
    end;
```

The Routines:

<u>stack(token)</u>	pushes the token on the stack
<u>getnext(statement)</u>	accesses the next token of the statement, in right to left order.
<u>backup(statement)</u>	backs up the statement pointer by one token to the right.
<u>type(token)</u>	return 1 for an "0", i.e. for scalar, array, IOD 2 for an operator 3 for a phantom 4 for a command, rightbrace, NOOP
<u>pop</u>	pops token from stack
<u>group(operator)</u>	returns 0 for an rdelim 1 for an execop 2 for numeric operators
<u>defer(OPR,AR)</u>	pushes on the stack the contents of OPR and AR.
<u>lookup(token)</u>	looks up the appropriate phantom table (local, global) and returns the entry (see fig. 4.6).
<u>phantomtype(token)</u>	returns 1 for S,AP, IOD 2 for UV 3 for FC 4 for PM
<u>argnumber(token)</u>	returns 0,1 or 2 according to the number of arguments of the function.
<u>resultnumber(token)</u>	returns 0 or 1 according to whether or not the function has an explicitresult.
<u>reset(linenum)</u>	performs the branch, i.e. resets the linenum to the appropriate integer value or causes exit to freturn, (functionreturn).

<u>descriptor(token)</u>	returns descriptor of token as string of characters
<u>field(token)</u>	returns right field of token as string
<u>result(AL,OPR,AR)</u>	executes the operation specified by OPR if monadic, the operand is in AR if dyadic, the operands are in AL and AR
<u>printAR</u>	prints contents of AR
<u>getnextline</u>	fetches next line( = statement) If no next line, then it exits to freturn.

An example of the operation of the interpreter, and its parser, is now given.

The statement to be parsed is:

(LD)                      (3 + B) + ÷ 1

where B has the value 2 and the LD has been introduced to the left of the statement by the translator (see Section 1 of Chapter III).

For clarity, only changes in the contents of registers or in the state are indicated. The operation of the parser is traced by using the Algol labels. In second reading, this can be compared to the microprogrammed interpreter (Section 6 of Chapter IV).



FIGURE 2.6 - OPERATION OF THE APL PARSER

LABEL	STATE	TOKEN	AL	OPR	AR	STACK
startline						LN
clear	0					
parsenext		1				
decypher						
operandtype	1		1		1	
parsenext		÷				
decypher						
operatortype	2			÷		
parsenext		+				
decypher						
operatortype						
monadic						
operate	1				1	
parsenext		+				
decypher						
operatortype	2			+		
parsenext		)				
rdelim						
clear	0					) + 1 LN
parsenext		B				
decypher						
phiprocessor		2				
operandtype	1		2		2	
parsenext		+				
decypher						
operatortype	2			+		
parsenext		3				
decypher						

FIGURE 2. 6 - OPERATION OF THE APL PARSER  
(Continued)

LABEL	STATE	TOKEN	AL	OPR	AR	STACK
operandtype			3			
dyadic						
operate	1				5	
parsenext		(				
decypher						
execop						+ 1 LN
lparen						
unstack		+				
endtest			5	+	1	LN
dyadic						
operate	1				6	
parsenext		LD				
decypher						
operatortype						
execop						
LD						
endline						
printAR						-

## 7. Semantic Auto-Specification

Some of the previous notions lead to a conjecture which, in order that it not be subject to too rigorous scrutiny, will be referred to as Zaks' Prejudice. At present, its formulation is too imprecise to support a rigorous proof.

Zaks' Prejudice: Syntactic simplicity implies terse auto-specification of semantically complex operations.

The syntax of a language defines its structure, while its semantics define its meaning or its transformation properties. A grammar, i.e., the syntactic specification of a language, often has to be changed to fit into the semantics. Extension or redefinition of semantics may thus prove to be particularly painful. Auto-specification refers to the specification of a new operation using exclusively previously defined operations. It implies an extension capability and permits redefinition to the extent of the auto-specified operations, as opposed to the basic set used for this specification.

The above prejudice is applied to an NPL. The language is defined with a very simple syntax and uses a uniform class of operators, the function-types. There is in particular no difference between a predefined function, or "operator" and a user-defined function, or "defined function". The semantic action associated with each "f" is thus completely independent of the syntax.

Relegating the syntax to a background position permits shifting all the emphasis on the semantics, i.e., the existence and creation of data objects with the methods to transform or manipulate them. In fact, once a basic spanning set of operators ("f" types) has been defined either by word-description or hardware-implementation, the NPL becomes an elegant and powerful tool for its own semantic specification and all remaining operators can be described in terms of the ones previously described (bootstrapping). This property has been detected intuitively for a long time already in the case of Iverson's language, which has been used primarily for the semantic description of computation algorithms. Practically, this feature of NPL programming languages allows the specification of most APL operators as software APL functions utilizing a basic spanning set of microprogrammed operators. Listings of APL operators expressed as APL functions appear in Appendix G.

# **CHAPTER        3**

## **TRANSLATION FROM EXTERNAL TO INTERNAL REPRESENTATION**

1. Internal APL Representation
2. Phantoms
3. Syntax Descriptors
4. Static Structures
5. The Translator

## 1. Internal APL Representation

Two general approaches can be considered for the interpretation of a high-level language: direct interpretation of the external symbolic form or transformation into an intermediate representation, followed by execution. It is shown here that an internal representation achieves an optimum for APL within its interactive environment. The design considerations for this internal APL representation are then presented.

In order to establish the respective merits of interpretation methods, the essential constraints on an interactive APL system should be pointed out.

- The system must be efficient. The requirement for fast interaction implies that lengthy delays should always be avoided.
- APL's naming rules preclude the identification of the syntactic type of identifiers or operators before the whole program has been defined (more on this topic in sections 2 and 3).
- Editing capabilities permit the selective redefinition of portions of a program during function suspension.

Direct interpretation of the external form has the following advantages: it uses a minimum amount of storage, since no intermediate representation needs to be stored, facilitates symbolic tracing and diagnostics and is simple to implement. However, the drawbacks of this approach are significant.

- All syntax analysis is performed at the latest point in time, i.e. execution time, when a large portion of it could be effected sooner (see below), at no penalty during execution.

\* Lexicographical analysis and data compaction must be carried on in a repetitive manner for any implicit or explicit iteration, thus carrying to a limit the inefficiency inherent to an interpreter.

In view of the inefficiency of this method, the necessity for an assembly phase becomes apparent. In this phase, all external symbols are encoded into a compacted internal token representation. The advantages of this internal representation are clear. Lexicographical analysis has been carried out with the associated building of symbol tables.

The timing for the start of this phase can in principle lie anywhere between type-in and execution.

In an interactive system such as APL, the following considerations must be taken into account in order to determine phase timing. It is critical both for user convenience and system efficiency to perform as much analysis as possible at the earliest point in time. For the user's convenience in an interactive environment, it is important to detect most lexicographical errors at the time the line is typed rather than delay the diagnostic. Most obvious errors can in this way be effectively detected and corrected. For computations during this period, response-time requirements are down to human typing speed and the system is not compute bound for the user. For system efficiency, the type-in period should therefore be used for intermediate computations.

On the other hand, performing this "translation" at type-in time introduces a limit on the extent to which the syntax analysis can be carried out. It has already been noted that APL's syntax

is not completely defined at that time. Identifier types and operator types are not resolved. This final resolution can be carried out either at the end of program definition, in the form of fix-up passes or at execution time. These two possibilities are now examined.

Fix-up passes at the end of program definition can be performed either at the end of each function definition or/and at the end of complete program definition, i.e., at the time execution is requested. In the case of editing during function suspension, a complete pass throughout the entire workspace becomes necessary. The absence of block structure in APL does not limit the scope of a global identifier, which may appear anywhere in the workspace. A fix-up pass will therefore generate a significant delay, intolerable in any interactive environment. Furthermore, the added efficiency gained by such an approach will be shown to be minimal in view of other methods.

The alternative approach consists of performing the remaining syntax resolution at execution time, i.e., when all the necessary information is available. The Phantom Materialization process is described in section 2 and the operator type determination in IV-6. The cost of this approach is nil in the case a Phantom materializes into a variable and is one PMS indirection (2 memory accesses) in the case of a function call or a label. Further, it allows labels to be treated like any local name and to be reassigned a value. In the case of operator type, the cost is nil in the case of a dyadic operator and of the order of a few micro-instructions, in the case of a monadic one (the following token found by the look ahead of the



parser is stored in a fast register and does not require a new memory access to be retrieved at completion of the monadic operation).

In summary, the need for an intermediate representation in an interactive APL environment has been established. The possible extent of syntax analysis which can be effected during this phase is limited by APL's naming properties. In an interactive environment, it becomes necessary to defer further syntax resolution until execution time. However, in the case of a microprogrammed processor, this analysis can be carried out at practically no time penalty.

Another benefit of this approach is the fact that the translation process is kept very simple and a one-one correspondence can be maintained between the symbolic external APL program and its internal representation. In the case of a severely limited memory system, it would be feasible to keep only the internal representation with its associated symbol table and regenerate simply the external form from it.

## 2. Phantoms

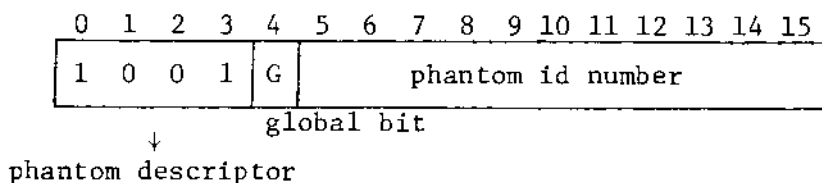
Every APL name or "identifier" may assume dynamically a number of distinct syntactic roles, such as: label, global variable, local variable, function. This "facility" for the use of synonymous identifiers introduces a significant complication in the translation process, since the syntactic type of an identifier cannot be determined at the time it is typed in.

A strategy therefore has to be devised in order to permit line translating and editing without major fix-up passes and later to determine dynamically the syntactic type of identifiers: all identifiers within

the system are uniformly translated into a special token: the "phantom", and receive a phantom identification number.

APL functions have been extended here in order to permit the use of optional parameters (see Chapter IV, section 2) and the significance of local and global identifiers can be stated simply: all variables appearing within the header of a function definition, and all labels appearing within the corresponding function body are local to the function. All other identifiers appearing within the function body are global. As a result, only global variables and those variables which are transmitted explicitly as parameters may become accessible to a subsequently called function.

One more piece of syntactic information is therefore directly available at input-time and is embodied within the phantom token in the form of a global/local bit:



$$G = \begin{cases} 1 & \text{when global} \\ \emptyset & \text{when local} \end{cases}$$

Figure 3.1 - A Phantom Token

The phantom id number ( $2^{11} = 2048$  entries) constitutes an index to the external symbol table on one hand and to the global or local phantom-tables to be introduced in Chapter IV. A phantom will materialize into a function call, scalar, array, null, parameter or undefined value (see Figure 4.6).

### 3. Syntax Descriptors

The choice of descriptors for the Program Strings has been dominated by APL syntax characteristics. Their physical distribution and ordering are naturally a function of the order-code and word-size available on the processor (16-bits in this case).

The Program Strings descriptors are organized in two groups:

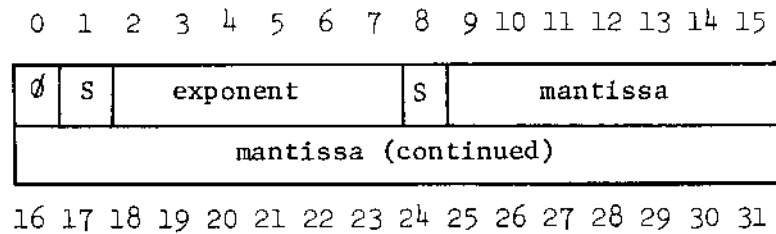
Group 1: 1 bit descriptor field for scalars which minimizes storage requirements for the descriptor.

Group 2: 4 bit main-descriptor field which represents the other three syntactic types: Array, Phantom, Operator and the facilities added to APL (NOOP, IOD, CO, RB, UN).

The details of individual field assignments are given below:



Figure 3.3 - Scalar (S)

Scalar:

bit 0 is the scalar descriptor.

bits 1-7 are the exponent in two's complement.

bits 8-31 are the mantissa in two's complement.

Some notes about two's complement:

- $-A$  is the complement of  $A+1$ .
- in a negative integer, 1's may be discarded at the left and  
0's at the right.
- in a positive integer, 0's may be discarded at the left.
- zero is represented by all zeroes and this is the only representation for zero.
- all even numbers whether positive or negative end in 0,  
odds in 1.

The one-bit format for the scalar descriptor allows maximum data transit.

The first bit of the exponent (bit 1) is the sign bit: 0 for positive, 1 for negative. The exponent may thus range from  $-2^6$  to  $+2^6-1$  or [1000000, 0111111].

The 7-bit exponent takes advantage of the ease in manipulating a byte in this machine as a unit for shifting and testing.

The mantissa has been limited to 24 bits because of the restriction on the size of floating-point routines that may reside within the 2K of ROM and on the number of hardware registers available.

When normalized, the first bit of the mantissa is the sign bit. The floating point is between bits 8 and 9 and the range of the mantissa is thus

•  $2^{-1}$  to  $1-2^{-23}$  for positive numbers or  $[0.100\dots0, 0.111\dots1]$ .

Because of the normalization, a '1' must appear after the floating point.

•  $-1$  to  $-2^{-23}$  for negative numbers of  $[1.000\dots0, 1.111\dots1]$ .

The range of the mantissa is therefore  $-1$  to  $1-2^{-23}$ .

Internal APL: The scalar appears with its two words reversed, allowing descriptor recognition during the right to left scan.

## Program Strings Descriptors

Figure 3.4 - Literal array (always a Vector)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	0	T	0	0	0	0	0	0	0	0	0	0	1
size															
0															
number of elements															
number of elements for dimension 1															
elements in															
row-major order															

APL Null is:

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Literal array:

The representation of a Program Strings Array (always a literal vector) is consistent with the representation of arrays throughout the system. For the general representation, see figure .

Figure 3.5 - Phantom (PH)

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

1	0	0	1	G	phantom number										
---	---	---	---	---	----------------	--	--	--	--	--	--	--	--	--	--

Phantom:

G is global bit (when 1) or local (when 0). Phantoms are used for the uniform representation of identifiers. The phantom #  $2^{11} = 2048$  entries) constitutes an index to a symbol table on one hand and to the global or local operand address table on the other hand. A table look-up identifies the phantom as function call, scalar, array, null, IOD, parameter, or undefined (see Figure 4.6 - Phantom Materialization).



## Program Strings Descriptors

Figure 3.6 - Operator (OP)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	1	0	R	X	M	L	C	I	op code					

Operator:

bits 0-3: OP descriptor

bit 4: (R) Right delimiter if on

bit 5: (X) execop if on

bit 6: (M) mixedop if on

bit 7: (L) logical if on

bit 8: (C) compare if on

bit 9: (I) indexable if on

The Right delimiters (RDELIM)

They are: RPAREN ) opcode #

RBRACKET ] opcode #

An RDELIM starts a new expression level and causes evaluation of any phantom which has to be deferred (and appears in the AR register of this implementation).

The Execution Operators (Execops)

They are:

opcode

BRANCH	→	0
LBRACE	{	1
LPAREN	(	2
LBRACKET	[	3
LD	LD	4
ASSIGN	←	5
SEMICLN	;	6

An execop closes an expression level and therefore forces evaluation on its right.

The M, L, C, I, fields:

See Appendix F - APL operators.

## Program Strings Descriptors

Figure 3.7 - NOOP

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

1	0	1	1												
---	---	---	---	--	--	--	--	--	--	--	--	--	--	--	--

This NOOP is used in various situations by the translator to insert markers (for comments especially) invisible to the parser.

Parser: the descriptor is ignored and parse proceeds. This descriptor should not appear during execution.

Figure 3.8 - Input Output Device (IOD)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	0	C	device number										

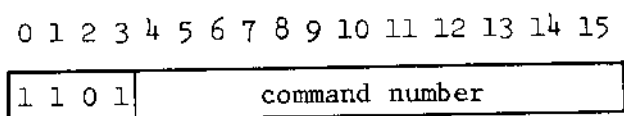
This special operand type is used for IO operations (quad, quote-quad).

C = 1 for character mode

Ø for line mode

## Program Strings Descriptors

Figure 3.9 - Command (CO)



bits 0 - 3 : command descriptor.

bits 4 - 15 : code for command number.

Commands are distinguished from normal operators in that they logically are addressed to the operating system.

## Program Strings Descriptors

Figure 3.10 - Right Brace (RB)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	0	parameter displacement											

The parameter displacement =  $2 \times (\text{parameters number}) + 1$  permits the immediate retrieval of the phantom associated with the parameter list and permits immediate syntax verification, without having to defer the state of the parser during the scanning of the parameter list, thus reserving the possibility of passing expressions as parameters. The phantom associated with the parameter list must naturally materialize into a function call. (See section 5.)

#### 4. Static Structures

The static structures are defined as being those which can be altered in Calculator Mode only. They include the Symbol Tables and the Program Strings. The only static structure accessible to the Interpreter proper is the Program Strings structure which contains the internal representation of all the user's functions.

As can be seen on figure 3.11 below, this structure consists of four entities:

-- The function table contains a four-word entry for each defined function. The function identification number is used as an index to the table entry. Its first word indicates the existence of a possible result (R), the number of arguments (AA) and the number of possible parameters. The second one carries a count of the total number of local phantoms, to be used at the time of the call in the construction of the local Phantom Materialization Segment (PMS). The third one points to word 0 of the Line Table, while the fourth points to the Label Table, if any.

-- Labels are stored individually in a Label Table and get loaded dynamically within the Phantom Segment at the time of the call, thus effecting label value initialization. In order to permit convenient line-editing, each label has a phantom number associated with its two-word floated value. Naturally, at the cost of a complete function retranslation after line editing, it is possible to eliminate the phantom number entry of the Label Table, since these numbers can then be guaranteed to be allocated consecutively

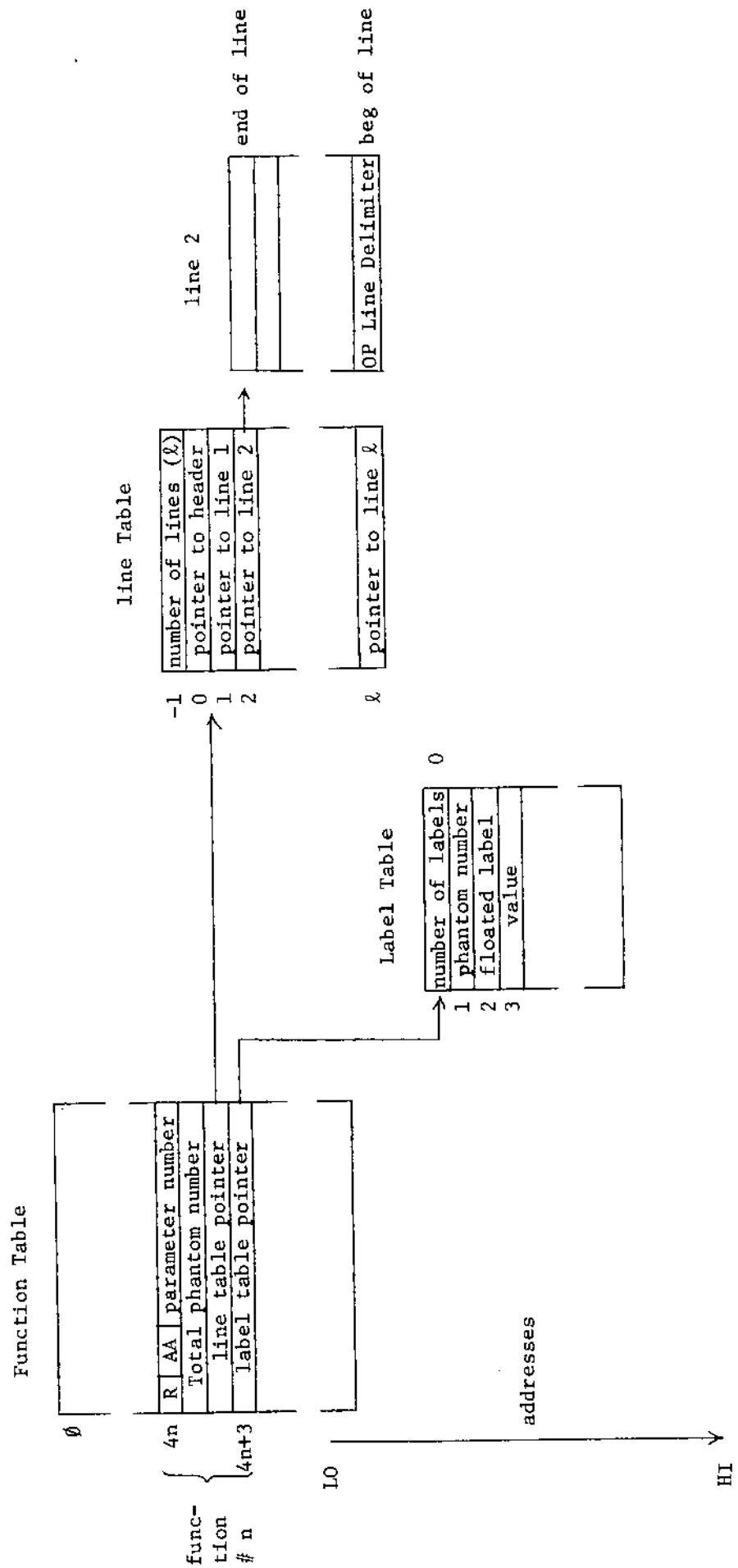
by the Translator and will be grouped at the top of the PMS.

-- The Line Table is equipped with a count of the number of lines, used before any branch or sequencing to a consecutive line, and one line pointer per word. These entries point directly to the last token of the line, since execution in APL must proceed from right to left.

-- Finally lines are stored high to low, so that their vertical representation exposes the end-of-line first. The beginning of line is delimited by a special operator, the line delimiter, supplied by the Translator.



Figure 3.11 - Program Strings: Static Structure for Functions



## 5. The Translator

The External strings presented by the user at the terminal must be converted to internal representation for execution. The software unit in charge of performing this transformation has been called the Translator.

The Translator essentially accepts APL statements and creates the Program Strings of Figure 3.11 along with symbol tables. Translation is effected on a line by line basis.

A lexicographical analyzer scans the APL line in left to right order and isolates the successive elements. The format of the tokens, and the associated descriptors are given in Figure 3.2.

Any number gets converted into a double-word token, the Scalar. Internally, there is no difference between integers and floating point numbers: they are represented by a scalar token. In other words, all integers are uniformly converted to floating form representation by the Translator. See Appendix H for examples. The Scalar is then placed in the translated line with its two words reversed. This permits recognition by the right to left scanner.

A number followed by one or more numbers in the input string denotes a (literal) vector in APL. The Translator therefore monitors every number for the possibility of a vector. Whenever two or more consecutive numbers are found, their scalar tokens are shipped in the usual fashion to the internal APL line being built. Upon detection of the last number in the external APL vector, a vector header is created by the Translator and placed to the right of the

scalar tokens. See Figure 3.4. The Translator has simply created the internal row-major-order representation of an external literal APL vector. This process did not require any stacking or back-up.

An identifier gets converted into a one-word phantom token with an indication of its type (local/global). The phantoms are discussed at length in section 2. In the case of a label (identified by the colon), the phantom is entered into the label table, along with the floated value of its line number. The phantom is not entered in the internal APL string. A label creates a new entry in the local phantom table.

In the case of a function header, a function name is created and a function call to the corresponding global phantom is entered in the GFRZ. The number of arguments, of parameters, and the eventual presence of an explicit result are determined. All the variables found in the function header, except the function name, are entered as local variables. The function name introduces a global variable.

An operator is generally directly encoded into a one-word operator token and appended to the right of the translated line. There are exceptions for special operators: a colon merely identifies a label, a quote identifies a vector, a quad or quote-quad are translated to an IOD token, operators denoting operating system functions are encoded as "command" tokens.

The right brace used to delimit a parameter list requires a special descriptor. In order to provide immediate syntax verification at execution time, its token is equipped with a "parameter

displacement" field. This displacement, added to the program pointer permits to access immediately the phantom which owns the parameter list. This permits to verify that the phantom is indeed a function call. This also permits to determine the action of the parser according to its state.

For example, in the case  $F\{P1;P2\} + A$ , the  $+$  can be monadic or dyadic according to whether  $F$  has one or zero argument. During right to left parsing, the right brace is scanned. Its deferment field permits to indirect to  $F$  and immediately determine the type of the operator according to the type of the function. In the case where  $F$  expects an argument, the  $+$  has been identified as a monadic operator and is executed. If  $F$  does not expect an argument, and provided it creates an explicit result, the  $\}$  causes stacking of the operator-operand pair  $(+ A)$ .

The Translator has been defined by the mapping from external APL representation to its internal "Program Strings" representation with the associated static structures. A flow chart of the Translator and the listings of its Snobol implementation are given in appendices D and E.

An example obtained on-line using a Snobol version illustrates now the structures developed so far. The APL code for the factorial function is typed in to the Translator as it appears within the frame in figure 3.12 (line numbers are furnished by the Translator). The Teletype representation used for special APL symbols is presented in appendix F.

Figure 3.12 - The factorial function - External APL

```

SNDFOL.
4.22
$READ FROM /TRANSLATOR.
$GO.
--OK.

```

```

FUNCTION START AT ... = 5
      02/20/72   1620:45

TYPE 'C' TO DELETE A LINE
TYPE 'END' WHEN FINISHED

```

```

META-APL  TRANSLATOR READY...

```

```

OK      SDEL Z ← FAC N :1

OK      11)  Z ← 1

OK      12)  I ← 0

OK      13)  L1 : I ← I + 1

OK      14)  $GOTO 15 IF I > N

OK      15)  Z ← Z * I

OK      16)  $GOTO L1

OK      17)  SDEL

OK      FAC 5

OK

```

END

```

EXECUTION REQUESTED

```

The Internal Strings and structures created by the Translator are presented in the following print-out. Several features can be observed on figure 3.13.

- The beginning of each external-APL line, i.e., the "end" of each internal APL line is marked with a line delimiter (122004) as indicated in the previous section.
- All multi-tokens elements have their external order reversed in order to permit efficient right to left parsing.
- The initial number of local phantoms is 3 at the time the header is typed in, but 4 at the time the function is closed ('\$DEL'), since there has been a label definition.
- Upon function closure, the associated structures are displayed: the local symbolic phantom table, the 4-word function table entry, the label table, the line table, the function call entry within the global zone (GFRZ - see Chapter IV, sections 3 and 4).

The reader interested in the actual implementation of the interpreter should familiarize himself with the descriptor notation by effecting the hand-translation of one or more lines, using the external APL representation and figure 3.2. The result should be identical to the listing below for the corresponding line.

```

*$DEL Z ← FAC N ; 1
* RESULT (R)      = 1
* ARG# (AA)       = 1
* PARAMETER #     = 2
* LOCALPHANTOMS # = 3
*-----+
* NEW FUNCTION : FAC !    FUNCTION # = 5
*-----*
*Z ← 1
CORE $ADDR,654:001501E,001502E,122005E,110001E,122004E;
*I ← 0
CORE $ADDR,655:001503E,001504E,122005E,110002E,122004E;
*L1 : I ← I + 1
CORE $ADDR,660:000500E,000501E,
      120000E,110002E,122005E,110002E,122004E;
*$GOTO $116 $101A I > N
CORE $ADDR,667:110005E,120017E,110002E,121023E,
      120002E,000505E,000506E,122005E,122004E;
*Z ← Z $116 I
CORE $ADDR,676:110002E,
      120002E,110001E,122005E,110001E,122004E;
*$GOTO L1
CORE $ADDR,682:110003E,122003E,122004E;
*$DEL
*LOCAL VARIABLES NUMBER IS 4
*
*****
*THE LOCAL PHANTOM TABLE:
* 0      = 110000 = N
* 1      = 110001 = Z
* 2      = 110002 = I
* 3      = 110003 = L1
*****
*
*LABELS NUMBER IS : 1
* LABEL 0 IS L1 = 3 (LINE #)
*
*LOAD GPR#2 WITH A NEW FUNCTION CALL TO FAC :
CORE $ADDR,4039:160006E,120000E;
*ENTRY IN FUNCTION TABLE:
CORE $ADDR,224:120000E,4E,1025E,1130E;
*CREATE LABEL TABLE :
CORE $ADDR,6E0:1E,3E,001140E,000000E;
*THE LINE TABLE IS :
CORE $ADDR,532: 6,0,650,655,660,667,676,682;
*
*FAC 5
CORE $ADDR,685:001520E,000000E,114000E,122004E;
*****
*THE GLOBAL PHANTOM TABLE :
* 0      = 114000 = FAC
*****

```



# **CHAPTER 4**

## **DYNAMIC APL STRUCTURES**

### **THE INTERPRETER**

1. Introduction
2. A New Functional Structure
3. Dynamic Memory Structures
4. Function Reference Zone
5. Dynamic Array Management
6. The Interpreter

## 1. Introduction

The characterization of APL as a Nested Primitive Language in Chapter II has resulted in the construction of a finite-state automaton for a simple parser. The Internal APL representation has been generated by the Translator described in Chapter III and the Interpreter is defined as the unit in charge of executing these Program Strings. In this chapter, APL's run-time behavior is characterized in terms of dynamic memory structures, and a systematic organization of the Interpreter's Virtual Memory is evolved from this analysis.

The functional components thus constructed are presented at first individually, then dynamically as they come into existence during an actual interpreter activation.

## 2. A new functional structure

The knowledge of APL's function structure is essential to the understanding of the Interpreter. APL's mechanism is outlined first. The modified addressing definitions developed for Meta-APL are then described.

Unlike ALGOL, all APL functions are defined at the same global level (level 0 or program level). Once called, an APL function may reference or create a name at level 0. It may introduce new local variables, which are explicitly defined in its header, and reference former local variables introduced by a previously called function. A local variable thus introduced by FN is said to be made global (and therefore accessible) to all functions called during the activation of its owning function FN. These variables are accessible

to all the functions called by FN, but when FN returns, none will be transmitted back to the calling level: their effect will have been local to FN. An obvious consequence of this strategy is the fact that a local variable created by function FN remains alive, i.e. accessible, to all the functions called during FN's lifetime ("activation"). Such an implicit mode of communication may result in undetected naming conflicts, especially when library routines are used, which may introduce a flurry of spurious names.

Since local names take precedence over global ones, it is not possible to guarantee that such conflicts will not occur without explicit inspection of local names for conflicts.

These considerations led to the design of a different addressing scheme where the inter-function communication of local variables is handled explicitly, through a parameter list, rather than implicitly. In Meta-APL, local variables introduced by a function are strictly local to the creating level and are not accessible to lower levels unless explicitly specified as

parameters in the function header. The syntactic format appears in Appendix B. In short, a Meta-APL function appears as:

[<left argument>]<function>[{<parameter list>}][<right argument>]

where in addition to the two possible arguments, a parameter list of arbitrary length may be specified.

The second advantage of this approach is the facility introduced for multiple argument specification, a particularly important feature for library routines.

APL's access mechanism is indicated graphically below:

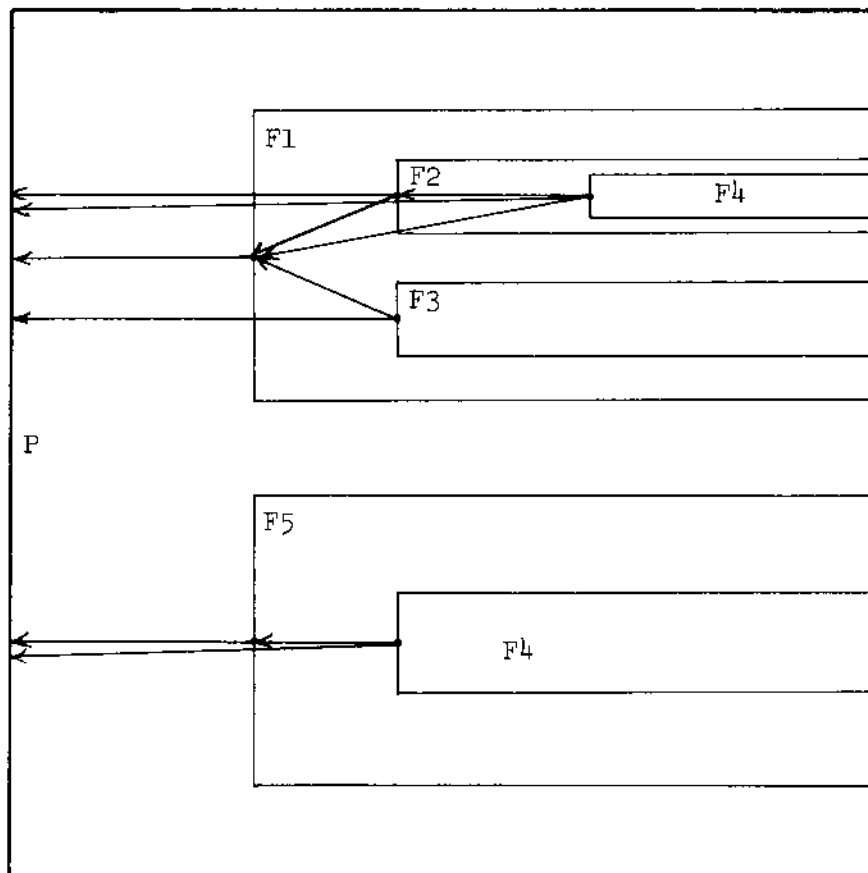


Figure 4.1 - APL's Inter-function Communication

where each solid arrow represents a valid implicit communication path between function levels. Meta-APL's inter-function communication mechanism is illustrated below:

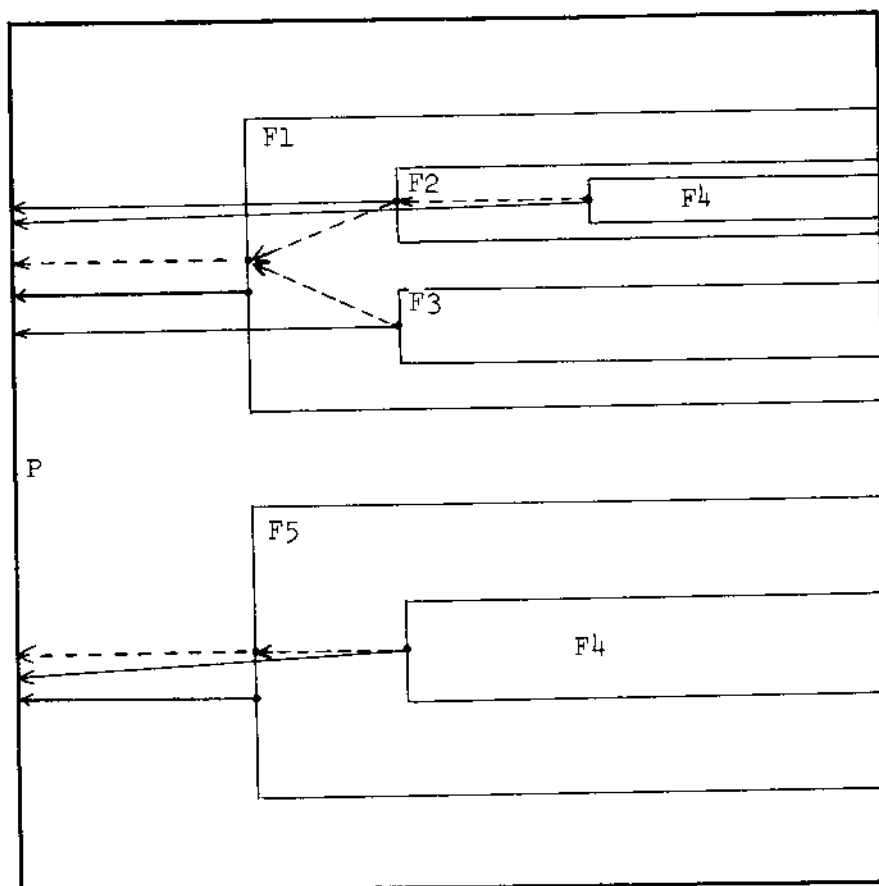


Figure 4.2 - Meta-APL's Inter-function Communication

where dotted arrows represent explicit parameter communications.

In summary, Meta-APL is the version of APL which has been implemented here. It is essentially identical to APL/360 with a modification and two extensions. The definition of global variables used in APL/360 has been modified in order to limit the scope of local arguments thereby eliminating a source of naming problems. APL has been extended with the use of parameters and the optional imbedding of any system command within an APL program.

### 3. Dynamic Memory Structures

The essential structure required for parsing APL has been introduced in Chapter II. The Stack permits the recursive deferment of operator-operand pairs upon entering a new expression level whenever a precedence-level is introduced by the parsing of a right-delimiter during the look-ahead phase. In particular, stacking an operator-operand pair identifies the deferred operator as dyadic and stores the evaluated operand. Although the operator and operand could be saved on separate stacks, there is no advantage in doing so if token-sizes for operands are uniform. In this system, all operand tokens are uniformly two-word in length when appearing within the Stack, thus permitting the simultaneous storage of operands and operators within a single Stack structure (see figure 4.5), where syntactic types can be simply segregated by this strict discipline (the token on top of the Stack is always a dyadic operator).

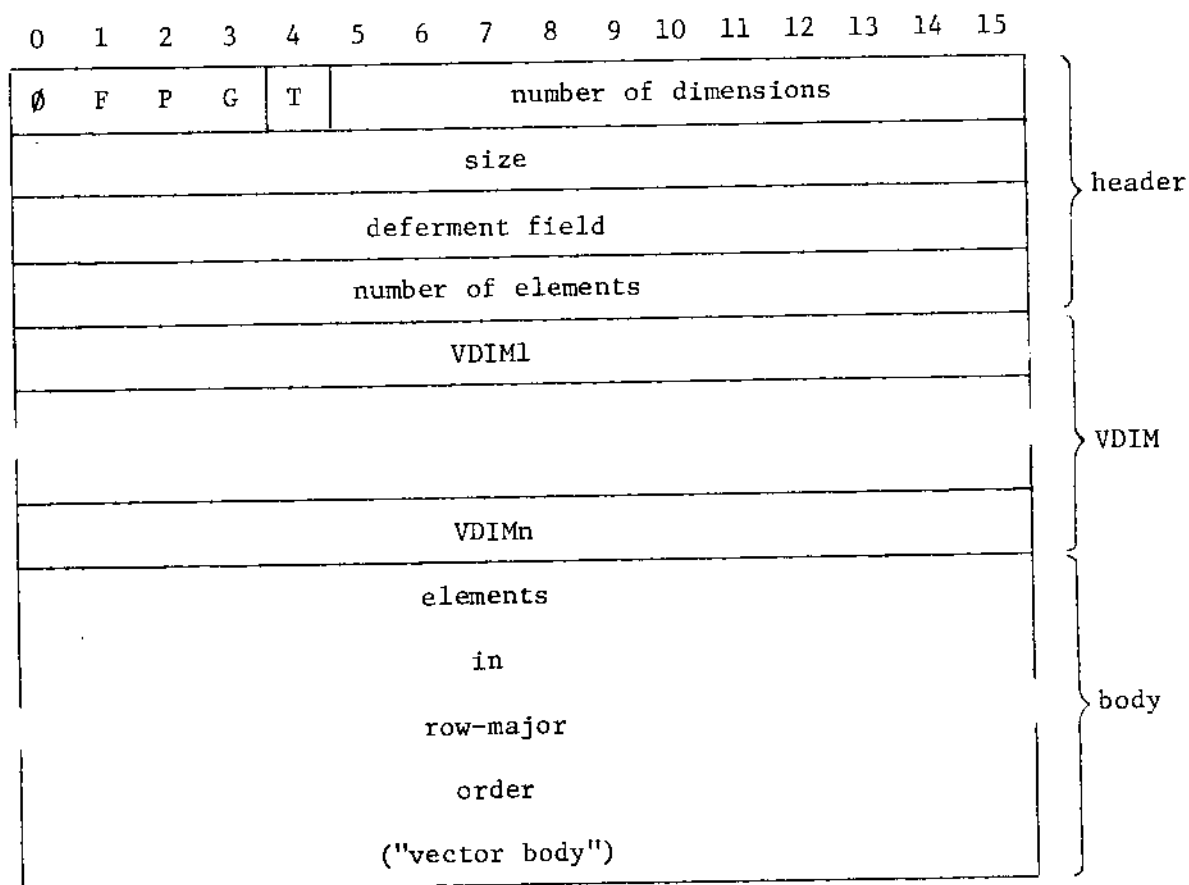
The second structure is brought about by APL's dynamic nesting resulting from the function mechanism. The Local Function Reference Zone appears upon function instantiation. This structure is synchronous with the Stack. The previous Stack remains constant while the LFRZ is active and resumes growing or shrinking only upon function return, i.e., after the LFRZ's disappearance. The LFRZ therefore does not necessitate a separate zone and can be stored on top of the existing stack. This structure is analyzed further in section 4.

A special case of the above structure is represented by the Global Zone, which corresponds to a "Program Call" and permits





Figure 4.4 - Array Block



APL Null is:

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

F is the Floating Bit used for storage allocation and deallocation

P is Previous block status 1 if empty

G is Actual block status 1 if empty (garbage bit)

T is the usual Type bit 1 if alpha (half words)

Figure 4.4 - Array Block (continued)

Word 1

bits 0 - 3 are the array descriptor

4 (T) represents the type:

-0 is a scalar array (double words)

-1 is an alphanumeric array (half words)

6 - 15 are the number of dimensions (up to 2048)

Word 2

size = total number of words of the array block

Word 3

Saved Stack field. Used for dynamic allocation and deallocation. Contains 0 initially, then the value of the Stack pointer if the array must be deferred.

Word 4

Number of elements for the array. Note that the number of elements is not directly related to the size, since in the case of alphanumerics, the last word may be only half-filled.

The number of elements =  $\prod_{i=1}^{i=n} \text{VDIM}(i)$

Words 5 through n+4

VDIM(i) is the number of elements for dimension i.

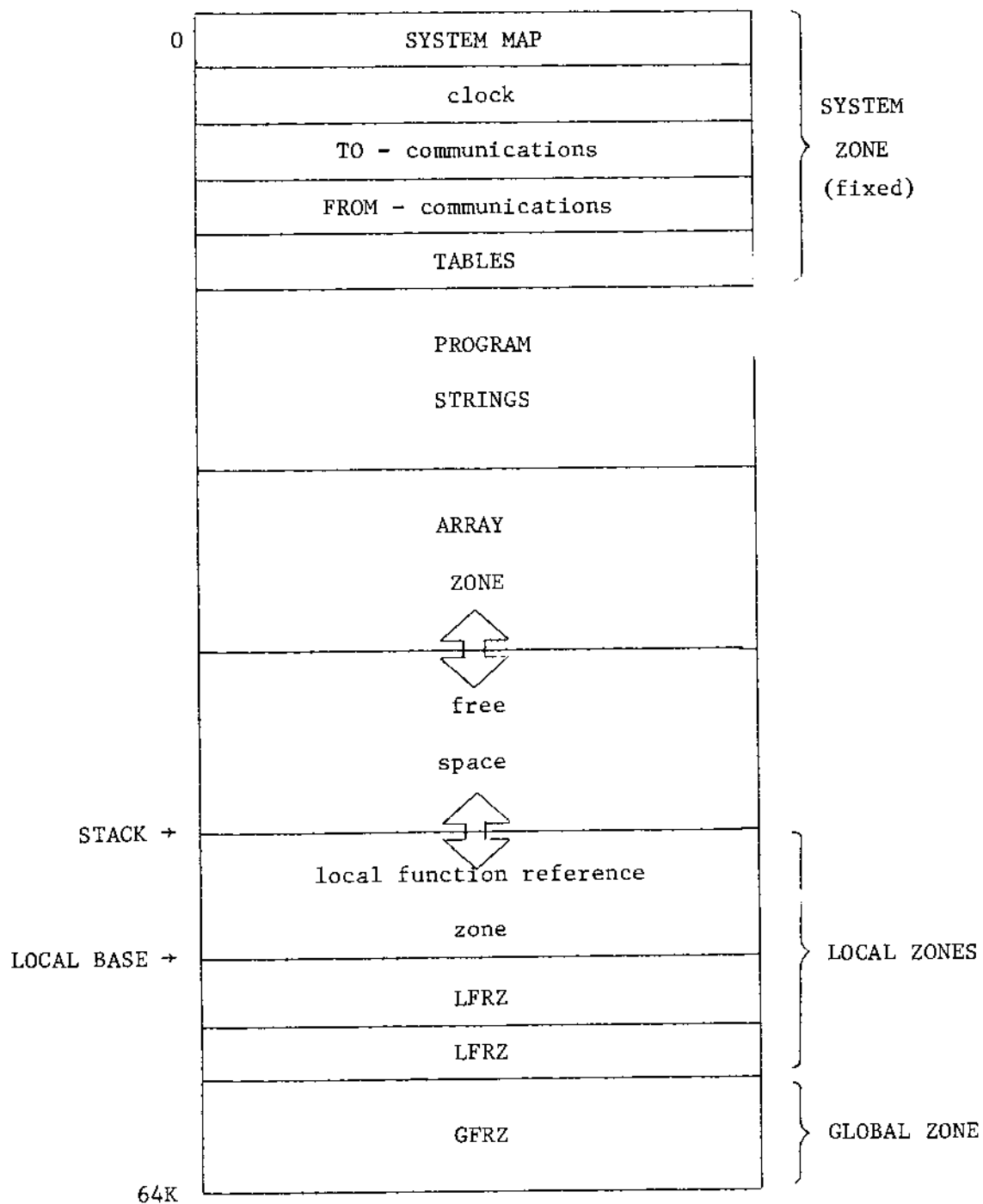
Remaining words: n+5 through end

Vector-body = elements of the vector in row major order.

the materialization of all global phantoms. This is the only zone guaranteed to be accessible at all times to parallel processes, if any, and the only one to contain the function-call tokens for the appropriate (global) phantoms. Finally, all arrays must be stored in a separate structure, the Array Zone, since their dynamic variations in size cannot be synchronized with any of the previous structures in view of APL's total absence of data-structure declarations. The management strategy for arrays is presented in section 5. Naturally a fixed zone must be reserved for operating system functions, including interprocessor communication in the case of a dual processor system: the System Zone.

This characterization of APL structures in terms of synchronization lends itself to a simple and elegant allocation where the Virtual Memory is simply allocated into fixed zones and two asynchronous structures growing and shrinking from opposite ends of storage. Interzone collision will then only occur when the process runs out of free space (see figure 4.5 below ).

Figure 4.5 - Virtual memory management



#### 4. Function Reference Zone

Operators and operands are deferred simultaneously and can therefore reside within the same Stack structure. Although function calls are syntactically comparable to operators, they need however carry with them two specialized substructures for linking and addressing. The important characteristic however is that these substructures (History Segment, Phantom Materialization Segment) are of fixed size and can therefore be imbedded within the preceding Stack structure.

The Local Function Reference Zone consists of three logical segments:

- The History Segment (HS) is created at the time of the function call and contains all the function characteristics (function number, result number, argument number, parameters number, line table pointer, number of lines) as well as the return pointers necessary for function return (former Program Pointer, former Local Base, Result Pointer). See figure 4.7.

- The Phantom Materialization Segment's name is self-explanatory: a double-word entry is reserved at function activation for each local phantom of the function. Initially, a few phantoms are materialized, namely the arguments, parameters and labels if any. All other entries contain a UV token (Undefined Variable). As assignments are effected within the function body, the remaining phantoms may materialize into new descriptor-values. Conversely, every phantom is evaluated by looking-up its value in the appropriate phantom table. All entries within the PMS are uniformly double-words.

In particular arrays are referenced by means of an Array Pointer entry (see figure 4.6). The size of the PMS therefore always remains strictly twice the number of local phantoms and is unaffected by dynamic size-changes of array-shaped variables, permitting its synchronous storage on the previous stack.

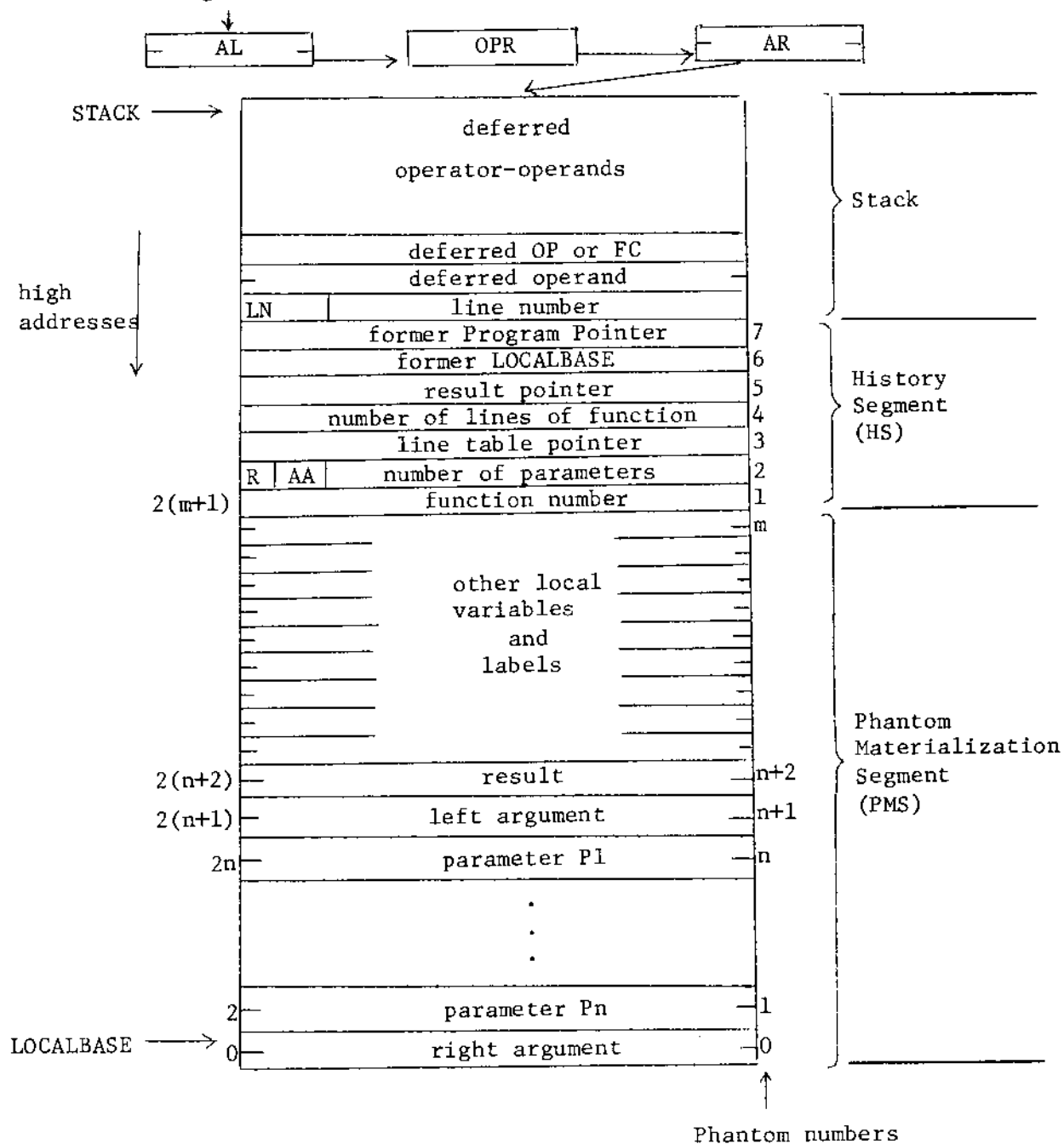
-- The third segment is the (local) Stack, used for recursively deferring the operator-operand pairs potentially encountered during the parsing of the function. The bottom of the Stack is identified by a Line Number descriptor which identifies the line within which deferment occurs. (For any instance of a function activation, there can be one and only one line in deferred state).

The detailed organization of the LFRZ appears in figure below.

Figure 4.6 - Phantom Materialization Segment Descriptors

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Scalar (S)	0	S			exponent				S							
Array Pointer (AP)	1	0	0	0	T											
Null (NULL)	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Unused	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	0	1												
Parameter (PM)	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
IO Device (IO)	1	0	1	1												
Process Parameter (PP)	1	1	0	0												
Unused	1	1	0	1												
Function call (FC)	1	1	1	0												
	R	A	A													
Undefined Variable (UV)	1	1	1	1	-	-	-	-	-	-	-	-	-	-	-	-
	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Figure 4.7 - Local Function Reference Zone (LFRZ)





## 5. Dynamic Array Management

### A. Introduction

In order to provide a powerful and convenient interactive user-level facility, APL operators allow the dynamic redefinition of data shapes during execution without the need for prior structural definitions. Hence the importance in any implementation of the efficiency of the space management strategy for optimizing the space-time occupancy of storage blocks.

The design of the array management will conform to the following axioms:

Axiom 1. No new block must be created which can occupy a previously allocated one.

Axiom 2. Every block must be released as soon as no longer referenced.

Axioms 1 and 2 minimize respectively the space and the time-occupancy.

Space allocation is now examined in the light of Axiom 1, and space deallocation in the light of Axiom 2.

All values considered in this section are array-shaped exclusively.

### B. Space Allocation

The two events which may necessitate the allocation of a new storage block are the application of an operator-type with array result and the deferment of an array value into the run-time Stack.

Two special and important subcases are the assignment operator and the literal arrays and they will be examined separately.

### B1. Result Arrays

Every result array created by the execution of a syntactic operator-type is by its nature temporary, the only way to attach a permanent value to a name being an assignment operation. These temporary array-shaped results created during a monadic or dyadic operation or function call require independent storage to be allocated if and only if none of the operands may be overwritten, as when operating on named variables rather than temporaries.

In the case of a scalar operator operating on numeric arrays, APL's linear order of execution on an array's row-major order representation permits the result array to occupy any of the argument arrays, if available. The result elements may be deposited in row-major order within the argument block during the array iteration loop, superceding only the element which has just been used.

### B2. Assignment

Let  $R \leftarrow B$ , where  $R = A$  prior to assignment. A new storage area has to be allocated for  $R$ 's new value if and only if neither  $B$  nor  $A$  are releasable. Furthermore, in the case where  $A$  is releasable, it must be at least as large as  $B$ . A formal definition of releasability will be given in (D34). Assignment is therefore a special case of the previous section.

In particular, a new block must be allocated everytime active pointers to  $A$  exist at assignment-time, as in the case:

$\langle \text{VARIABLE} \rangle \leftarrow \langle \text{VARIABLE} \rangle$

or in the case of an indexed assignment  $R[I] \leftarrow A$ , where a new block must be allocated for the result independently of any other release conditions.

### B3. Deferred Arrays

An array-shaped variable which gets deferred during line evaluation must have its deferred value protected against possible side-effects. In order to guarantee a strict right to left evaluation, deferred values must not be affected by any assignments effected during the deferment period. Such side-effects are caused by a new value being assigned to the variable before its deferred value gets used.

This effect occurs in two contexts: middle-of-line assignment and function-call involving an assignment to the variable.

```
An example:      A ← 15
                  (A ← 1 1 1 1 1) + A
                  2 3 4 5 6
```

The rightmost  $A$  on the second line references the value 15 and must not be affected by the subsequent middle-of-line assignment.

To guarantee a syntactically correct right-to-left execution, a variable is evaluated everytime it gets deferred. A simple method to guarantee correctness then consists of creating an explicit copy of the value at the time. This approach is correct, but not optimal. A copy need not be created until it becomes explicitly necessary at the time an assignment is effected to a deferred variable.

This implies the introduction of a deferred mode for such array-variables if Axiom 1 is to be satisfied and the "Floating" Strategy below describes this in detail.

#### B4. Literal Arrays

In the spirit of Axiom 1, literal arrays appearing in the Program String need not be explicitly transferred to a new block in the Array Zone at the time they are parsed, since editing a portion of a line where the literal resides invalids resumption of execution at that level.

Transferring literal arrays to the Array Zone at parsing time might appear as allowing a more uniform treatment of arrays types, but leads to a possible proliferation of copies in the case of iterative functions or recursions.

It will be seen that the "Floating Strategy" needs not distinguish between literal and other arrays and treats them uniformly. In particular, literal arrays will never be released. They belong to the Program Strings Zone.

#### C. Space Deallocation

Deallocation of one or more blocks may occur in the following contexts:

- 1) operator-type applied to array-shaped operands, including an output operation
- 2) assignment to a previous array-valued variable

#### C1. Post-operation release: $R = (A \text{ OP } B)$

Any, or both, of the array-shaped operands must be released if

they are no longer referenced.

In particular, temporary "result-arrays" created as the result of an intermediate computation may be released as soon as used in a subsequent computation, since no pointer to them is ever created.

It has already been noted that literal arrays encountered in the Program String are kept there and need not be released.

A deferred variable is still referenced as long as no new assignment has been performed to the variable and may not be released until then.

#### C2. Assignment to a previous array-valued variable: $R \leftarrow A$

No value assigned to a variable  $R$  may be released until either a new assignment is effected to  $R$  or the owning function returns.

However, a block may be released everytime an assignment is effected to a formerly array-valued variable which has not been deferred and is not a literal.

#### D. Floating Blocks

A simple algorithm is described first, which is guaranteed to release all blocks which ought to be released, but whose timing is not optimal, thus conflicting with Axiom 2.

The "floating" strategy of array management is then introduced, which optimizes both the space and time occupancy.

#### D1. A Simple Algorithm

Upon entry of each new function level, two cells are allocated for storage of the extremity-pointers of a "Scratch-List", initial

and end pointers. These pointers are initially zeroed.

While in the function level, each request for a new block results in the block being placed in the Scratch-List of the current level, unless it is an assignment to an outer block (parameter or global variable), in which case it is placed on the Scratch-List of the outer block.

At exit of the function level, this Scratch-List is simply released, either explicitly or by appending it to the Free-List.

The advantages of this algorithm are its obvious theoretical correctness in guaranteeing all allocated blocks to be released and its simplicity of implementation.

However, Axiom 2 is not satisfied. Blocks are not released as soon as no longer referenced, thus cluttering storage. This is not acceptable in a memory limited system and an optimized algorithm is now presented.

## D2. Floating Strategy

### D21. Optimization

Abrams in his thesis [Abrams, 1970] has pointed out that the literal evaluation of APL expressions may involve unnecessary intermediate computation. He therefore recommends deferring evaluation of operands and operators as long as possible ("drag-along") and points out that  $A+B+C+D$  may be evaluated element by element as:

for  $I \leftarrow 1$  step 1 until  $\rho A$  do

$T[I] \leftarrow A[I] + B[I] + C[I] + D[I]$

thus eliminating many unnecessary fetches and stores. Drag-along is applied to all array operands.

An example makes clear a limitation of this approach:

$$C \leftarrow A[B \leftarrow D] + B + A$$

$B + A$  should be evaluated by the time  $B \leftarrow D$  is evaluated or the results will be incorrect. Evaluation cannot be deferred in this instance.

The maximum limit is obtained by considering that evaluation of an operand may be deferred as long as it does not get modified. Modification can be detected since there is only one operator which has the effect of changing a value attached to an identifier, namely the assignment operator.

It can therefore be stated:

Evaluation of an operand may be deferred up to an assignment to the pending variable, or the end of the line, whichever is detected first.

This mechanism may be implemented by storing " $V^{\dagger}$ " where  $\dagger$  is an internal symbol indicating that  $V$ 's evaluation is pending.

As a consequence of this result, whenever a literal vector is encountered in the source string, there is no need to copy or transfer it until actually needed. A pointer to it is sufficient since no assignment can modify its value.

The "Floating" strategy derives its name from the "floating" attribute, indicated by a floating bit  $F$ , assigned to each array block.

$F = 1$  for a floating block, i.e., one which may be released once utilized.  $F$  is 0 for a named block, 1 otherwise.

$F$  gets turned on everytime a non-named block is created (a result array) or a block is no longer named (new assignment to a previously array-valued variable).

D22. Assignment

Everytime an assignment is performed, the variable is checked for a previous array value P, in which case P is "detached" and marked as floating. The variable will no longer reference this block, but the deferred pointers in the stack, if any, will.

D23. Deferment

Since a variable may be deferred an arbitrarily large number of times during the same line, such as:

$$(((\dots + A) + A) + A) + A$$

a deferred variable may not be explicitly released until all pointers to it have disappeared, i.e.,

- (1) A new assignment has been effected to the variable.
- (2) All references to it have been removed from the Stack.

In contrast with the possible "reference counter" technique, a "unique name" identification is used here by associating with each deferred array the corresponding Stack pointer at deferment-time.

A deferred array, once floated, may therefore be released if and only if :

$$\text{current Stack pointer} > \text{saved Stack pointer}$$

D34. Algorithm

The Release Condition R is:

$$R = 1 \text{ if } F = 1 \text{ and } \text{Stack} > \text{Savedstack}$$

$$R = 0 \text{ otherwise}$$



Note that in the case of a literal assigned to a variable, the literal array will never satisfy the Release Condition and is copied before the assignment is performed. Also, its floating bit can never be turned on and it will never be released. No special treatment is necessary for literal arrays.

The two attributes of an array block are presented in the table below in their various combinations.

array type	F bit	Deferment field
literal	0	---
temporary result	1	0 or SAVEDSTACK
non deferred variable	0	0
deferred variable - "fresh"	0	SAVEDSTACK
deferred variable - reassigned	1	SAVEDSTACK

Figure 4.8 - The Floating Bit

The alert reader will notice that the algorithm as presented guarantees a minimum number of array copies created as the result of Stack deferment or intermediate operations. In the case where the array value of a variable is directly assigned to another one, the algorithm will create a separate copy which could theoretically be avoided. This however does away with the traditional annoyances of a reference counter which has to take into account cases such as multiple redefinitions of the same variable:

A ← 25

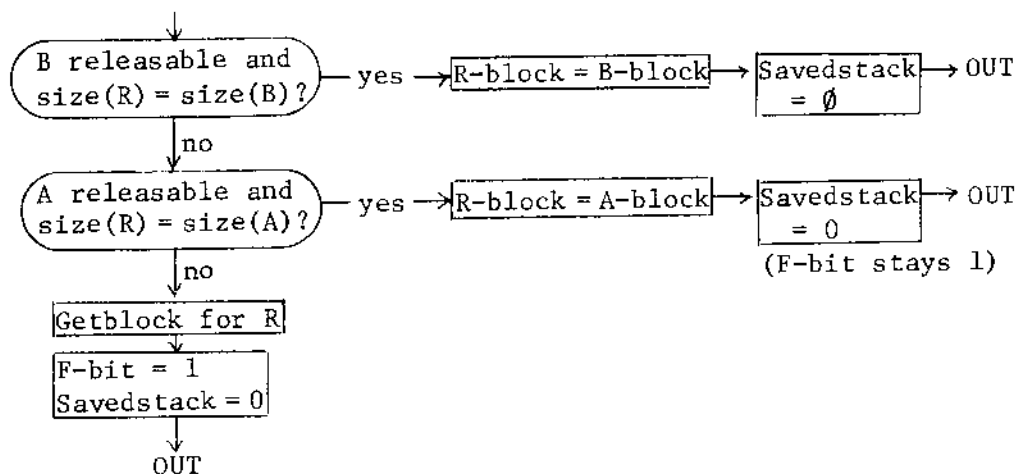
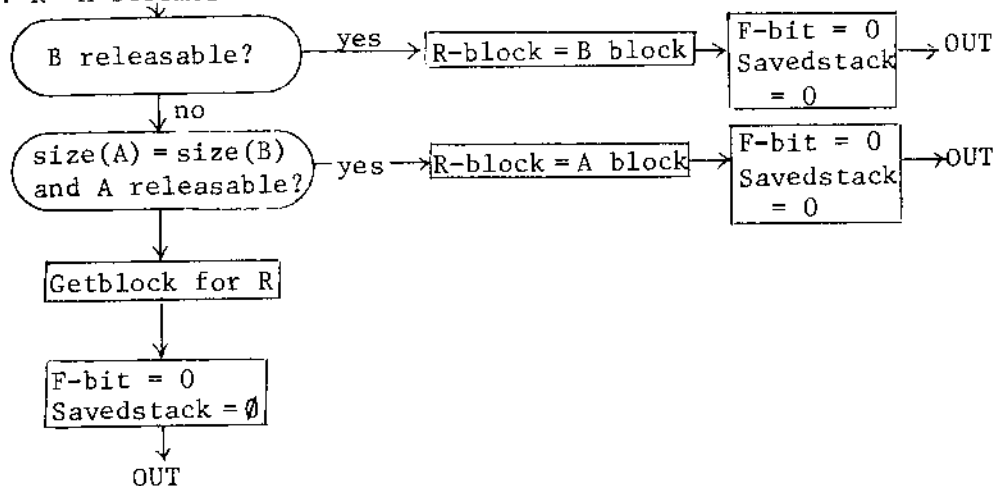
B ← A

. . .

A ← B

where the reference counter manager must recognize multiple definitions of the same variable in order not to increment the reference count indiscriminately. Such a facility can be provided for the salvation of the tormented optimizer by using the Savedstack field for the storage of: -(reference count). The floating algorithm would be unaffected by such a use.

Figure 4.9 - Storage Allocation

1 - Result Array:  $R = A$  or  $B$ 2 - Assignment:  $R = A$  becomes  $R \leftarrow B$ 

3 - Deferment

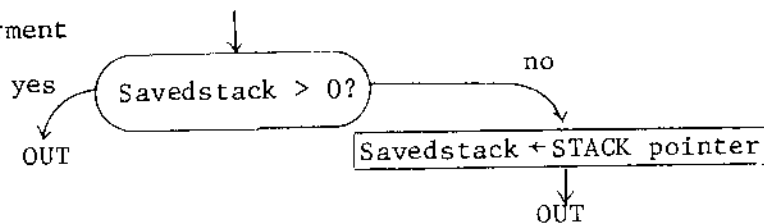
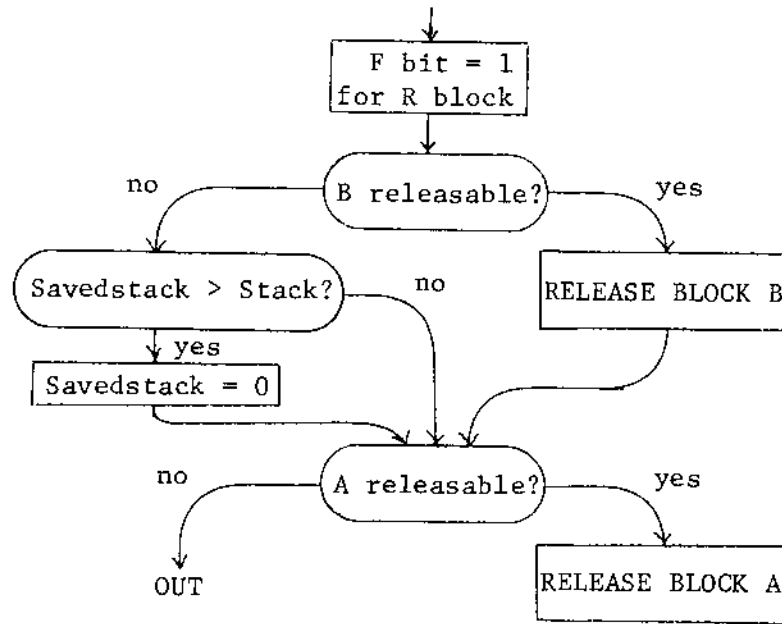


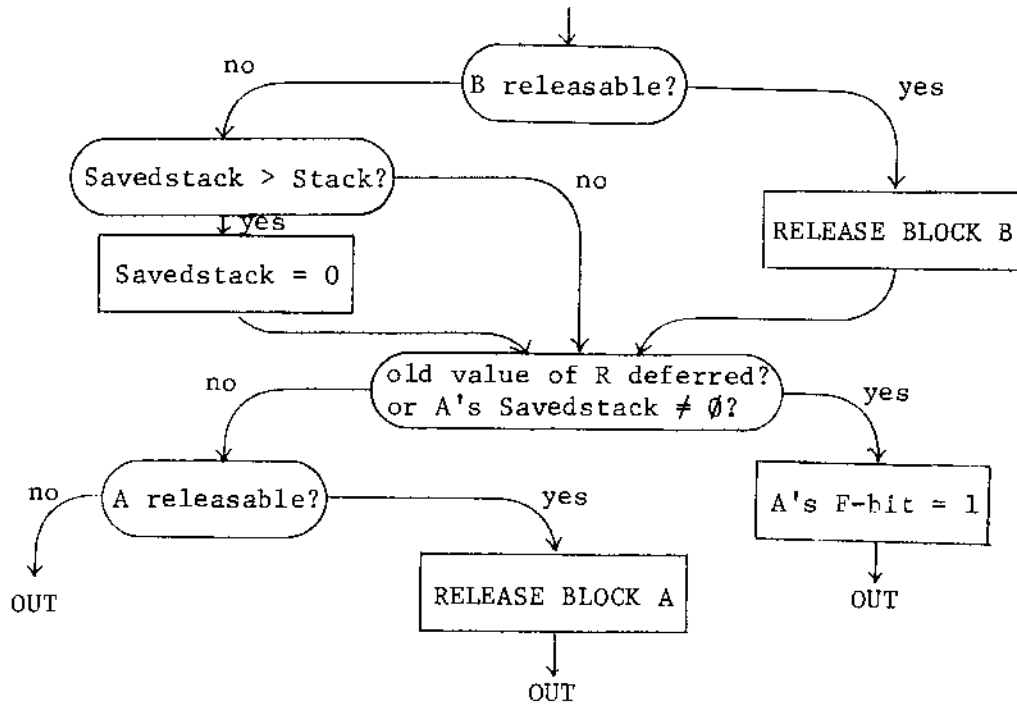
Figure 4.10 - Storage Deallocation

1 - After operation:  $R = A \text{ OP } B$

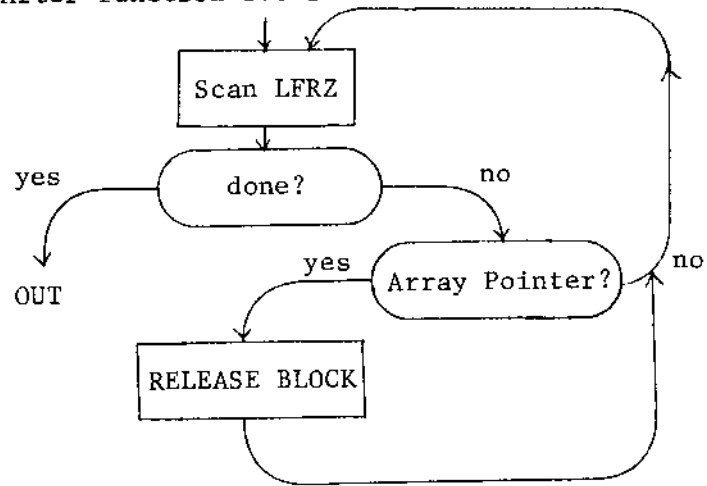


OUT

2 - After assignment:  $R = A$  becomes  $R \leftarrow B$



3 - After function return



## 6. The Interpreter

### A. Introduction

The concepts developed in this chapter have led to the design of a very terse interpreter for time-shared APL execution, capable of residing within the restricted control storage (2K) of an existing microprocessor (Digital Scientific's Meta 4 processor). The three-state parser developed in Chapter II scans the static structures of Chapter III for syntactic resolution, utilizing the dynamic structures of Chapter IV. The semantic execution routines for APL operators constitute the topic of a separate analysis (Chapter V).

In order to illustrate these concepts and to demonstrate visually the dynamic behavior of these structures, a prototype for the interpreter has been encoded for the Meta 4 processor and the results of a sample run are presented in the last section.

### B. An interpreter for time-shared APL

The hardware-independent run-time representation which has been developed permits the time-shared reentrant execution of the interpreter (firmware processor) by segregating the control program (firmware interpreter and fixed software operators) from the activation record (virtual space) associated with each process. Furthermore, the fixed positions of the Program Strings Zone and of the Global Zone for each process permit the system-wide sharing of function-bodies and inter-process communication at the global level via possible process parameters (as an APL extension for parallel processes). The actual implementation of the firmware interpreter onto the hardware Meta-4 processor, and the associated hardware-

considerations are presented in Chapter VI.

The initial virtual space prior to execution and after translation includes the program strings zone and the global zone, where the Phantom Materialization Segment's (PMS) entries for functions have been initialized. All other global entries are set to Undefined Variable and the global Stack is initiated on top of the PMS (initial STACK pointer). Caution: in this representation, addresses increase going down the page; low addresses are on top of high addresses.

During execution, the instantaneous description of an APL process can be effected in the following terms:

- 1 - The Virtual Memory Zones: System Zone, Program Strings Zone, Array Zone, Local Zones, Global Zone.
  - 2 - A set of semi-infinite tapes for input-output communication.
  - 3 - The process state-word which contains the set of virtual registers permitting the time-shared instantiation of the process on the virtual processor.
  - 4 - The address-mapping information establishing the correspondence between the virtual and the physical processors.
- The state-word of the process consists of a set of dedicated registers for process-wide pointers and a set of general-registers for module execution and intermodule communication.

The dedicated pointer-registers are:

PP: The Program Pointer, pointing to the next token to be parsed within the Program Strings.

LOCALBASE: the base pointer for the most recent Local Zone created, i.e., for the Local Phantom Materialization Segment.

STACK: the current pointer to the top entry of the current Stack level.

ARRAY ZONE: base and toppointers

FUNCTION TABLE BASE: self explanatory

The other registers are:

FLAGS: for storage of parser state and interpreter status.

LINK, RETURN: for return from nested subroutine calls.

ALØ and AL1 contain the left operand and ARØ, AR1 contain the right operand, while OPR holds the operator token for operator execution.

VMA and VMD are used for communication with the memory access routines: Virtual Memory Address and Virtual Memory Data.

Finally, about ten other registers are allocated for general purpose storage of various data or pointers during the execution of individual Interpreter routines. On a physical processor equipped with a sufficient number of hardware registers, most or all of these pointers can be directly implemented as a hardware register, the remainder being allocated out of fixed memory locations.

### C. APL semantics

The semantics of APL can be specified directly in the micro-code or at a higher level in terms of the dynamic changes of the instantaneous description during interpretive execution. The changes within the previous structures brought about by the various APL function-types (function calls, special operators, numeric operators) are now examined.

Function calls and special operators are considered in this



section, while the numeric operators (scalar and mixed APL operators), including indexing as a dyadic operator are the topic of Chapter V.

#### C1. Phantom Materialization

Upon identification of a Phantom within a Program String by the Parser, the  $\phi$ -Processor (see Chapter II, section 6) is called in order to materialize the Phantom into one of the PMS descriptor types (see figure 4.6).

The proper Phantom Materialization Segment (global / local) is selected by examining the global bit of the Phantom token and the PMS entry looked up, materializing the Phantom into its double-token value. A parameter requires an extra-iteration for complete materialization into a value or Undefined Variable.

Four types of final incarnation are possible: Undefined Variable (UV), Scalar (S), Array Pointer (AP), Input-Output Device (IOD), Function Call (FC).

A UV results in a syntax error when identified in any other position than on the left of an assignment arrow, while a Scalar or Array Pointer are deposited in the AL or AR registers, depending on the state of the parser, unless the state is 1 (syntax error: operator or function call expected). An IOD results in a call to the appropriate IO routine for evaluation (input) or output. Function calls are examined in the following section.

#### C2. Function Call

APL Function Calls within this system come in a variety of syntactic types: from 0 to 2 arguments and from 0 to n optional parameters. They may further either leave or not an explicit result

upon return. However, the fundamental semantic action associated with the function call is common and consists of the creation of a new Local Function Reference Zone (LFRZ).

The detailed procedure follows. The double-word function call is examined for syntactic validity in function of the state of the parser and according to the number of arguments and the existence of a possible result. The number of actual parameters, if any is checked against the number of formal parameters. Upon successful matching, two cases are distinguished: 2-argument function and other function.

A 2-argument function switches the parser to state 2 and causes parsing to resume, as in the case of a regular operator.

Any other function (0 or 1 argument) does not require any further parsing and interpretation proceeds. The new Program Materialization Segment is created, containing the eventual evaluated argument and/or the eventual parameter pointers. The floating-point representation of labels, if any, is obtained from the label table of the function and entered at the appropriate PMS location. Finally, all remaining PMS entries are initialized to a UV descriptor (Undefined Variables).

The History Segment for the function is then constructed (see figure 4.7) and contains the following information:

1. Program Pointer, pointing to the next Program Strings token to be parsed upon function return.
2. Localbase, pointing to the base of the Local Zone to be exposed upon return.

3. Result pointer, for the expeditious retrieval of the result if any.

4. The Number of Lines for the function, to be checked against any computed line number during function execution, and obtained from the function table entries.

5. The Line Table Pointer for the function, obtained from the function table entry.

6. A one-word entry for Result number, Arguments number, Parameters number.

7. The function number, for tracing, diagnostics and suspension.

The LFRZ has now been activated and execution of the new function proceeds after setting the line number to one.

A Function Return is effected in the following fashion. The PMS is scanned for AP entries and all releasable-blocks are garbage-collected (G-bit on). The History Segment is then used to restore the previous function level. The Program Pointer, Stack Pointer and Local Base are restored. The result, if any is accessed through the Result Pointer and placed in the AR (right argument) causing parsing to resume in state 1. In the absence of a result, the next token is verified to be an LD and NEXTLINE is called.

### C3. Operators

Upon detection of an operator token by the parser, the OPERATOR routine examines the descriptor for classification into one of three operator types: "right-delimiters", "executable operators" and "numeric operators."

Naturally, at each stage the appropriate syntactic checks

are made but are not described in detail here, in order to present a simple functional description.

The Right Delimiters are `) ] ;` and introduce a new expression level within the line, resulting in the evaluation of the AR, followed by the deferral of the OPR (operator) and AR (right argument) into the Stack. Parsing resumes in state 0.

Executable Operators are `( [ {` → LD (line delimiter) and close an expression level. They each cause a call to the appropriate specialized semantic routine:

1 - A branch verifies the next token to be an LD and examines its right argument; if Array, the first element is obtained. The resulting argument is verified to be a non-negative scalar and fixed. Zero causes a return. The fixed integer is compared to the number of lines for the function. An overflow causes a return, while a normal branch simply resets the line number and starts a new line (STARTLINE).

2 - A left brace simply verifies that a matching right brace has been found and parsing proceeds.

3 - A left Paren verifies that a matching right Paren appears on top of the stack, pops it and examines the next Stack entry. A Rightdelimiter causes parsing to resume in state 1 (PARSENEXT). Any other operator is a dyadic operator. AR is transferred to AL and 3 words are popped from the Stack into OPR (operator) and AR (right argument). The dyadic operator routine is called (BINOPER).

4 - A left bracket checks the top of the stack for `; or ]` and causes parsing to resume in state 2 (PARSENEXT).

5 - A line delimiter examines the top of the stack for an LN (line number created at the beginning of every line). An LN causes the AR to be printed, with return to NEXTLINE. In the absence of an LN, a ; must be found, causing the AR to be printed, with return to the beginning of this routine.

Numeric operators are distinguished as monadic or dyadic by the state of the parser. A numeric operator parsed in state 2 identifies a monadic operator (MONOPER) while one parsed in state 1 simply loads the OPR, sets the state to 2 and causes parsing to resume (PARSENEXT) unless it is an assignment (ASSIGN). Further details are found in Chapter V.

Most APL operator symbols may be used to specify monadic as well as dyadic operations.

Determining the type of operators is not possible in a one-line or even a one-function translating environment. In view of forward definitions, the syntactic type of operators can only be determined after all function definitions have been closed. The only two options possible are therefore a complete pass over the entire program with appropriate local retranslation or run-time analysis. As has already been indicated in the case of phantoms, a complete fix-up pass prior to execution is unacceptable and type analysis of operators must be performed dynamically. The three state parser capable of such identification has been constructed in Chapter I.

An example:  $C \ A+B$  is a valid statement with the following possibilities:

- B is a variable or niladic function with result
- A is a monadic function  $\Rightarrow +$  is monadic and  $(+B)$  is the argument of A
- or A is a variable or niladic function with result
  - $\Rightarrow +$  is dyadic
  - $\Rightarrow C$  must be a monadic function whose argument is  $(A+B)$
- or A is a dyadic function  $\Rightarrow +$  is monadic
  - $\Rightarrow C$  must be a variable or niladic function with result.

The syntactic type of operators is therefore resolved by the look-ahead unit of the parser which upon recognition of the operator analyzes the next token on its left.

#### D. A Practical Illustration

An actual run of the interpreter is presented in order to demonstrate the previous structure.

The interpreter has been microcoded and tested for verification only and must not be considered to represent in complete detail the final form to be used in an actual system. This microprogrammed interpreter runs on a Meta-4 simulator on an XDS 940 time-sharing system. This simulator has been verified to emulate exactly the hardware processor. The listings are supplied in appendix M.

In order to facilitate the interpretation of numerical results, a correspondence table for floating point numbers is also supplied in appendix H.

In this example, the recursive version of the factorial function is executed. Its external specification is given in figure 4.11,

translated into a set of static structures (figure 4.12) which is in turn executed by the Interpreter. A number of breakpoints have been set within the Interpreter in order to provide a visible trace of important routine entries. The proceed command ([PR;) is issued after each breakpoint. The contents of memory are examined with the "[CE MADDR,....,....;" command. In particular, due to a core limitation within the simulator, the Interpreter runs within a virtual 65K address space, even though the Meta-4 simulator supplies only 4K of actual core memory. Core location 7777B (where B indicates octal) thus denotes 4K "physical" or 65K "virtual".

The Interpreter starts in calculator mode (CALCMODE) and executes the function call, resulting in the creation of the first LFRZ (figure 4.13) where the History Segment (HS1), the Phantom Materialization Segment (PMS1) and the Stack are easily identified. Subsequent recursive calls of the FAC function result in activations of levels 2, 3 and 4. Level 4 finally evaluates into value 1, explicit result which is transmitted to the previous result. Function levels are then peeled off, transmitting the final result 6 to the calling level and causing a printout (PRINTAR) of the value 6.

Figure 4.11 - The factorial: Translator input

BASE = 650 ? ... = 650

TODAY ' S DATE IS : 01/33/72  
TIME OF THIS RUN : 1357:14

TYPE '@' TO DELETE A LINE  
TYPE 'END' WHEN FINISHED

META-APL TRANSLATOR READY...

OK	\$DEL Z ← FAC N
OK	[1] \$GOTO 4 \$TIM \$IOTA N = 0
OK	[2] Z ← N \$TIM FAC N - 1
OK	[3] \$GOTO 0
OK	[4] Z ← 1
OK	[5] \$DEL
OK	FAC 3
OK	FAC 7
OK	FAC 7
OK	FAC 1

END



```

EXECUTION REQUESTED?      NO
INTERNAL APL STRINGS READY ON :TEST*:*:

*$DEL 2 ← FAC N
*RESULT (R) = 1
* ARG# (AA) = 1
* PARAMETER # = 0
* LOCALPHANTOMS # = 2
*-----*
*NEW FUNCTION : FAC !
*-----*
*$GOTO 4 $TIM $IOTA N = 0
      CORE MADDR,650:000000B,000000B,120220B,110000B,121023B,
      120002B,001500B,000000B,122000B,122004B;
*Z ← N $TIM FAC N - 1
      CORE MADDR,660:000500B,000000B,120001B,110000B,114000B,
      120002B,110000B,122005B,110001B,122004B;
*$GOTO 0
      CORE MADDR,670:000000B,000000B,122000B,122004B;
*Z ← 1
      CORE MADDR,674:000500B,000000B,122005B,110001B,122004B;
*$DEL
*LOCAL VARIABLES NUMBER IS 2
*
*****
*THE LOCAL PHANTOM TABLE:
* 0      = 110000 = N
* 1      = 110001 = Z
*****
*
*LABELS NUMBER IS : 0
*
*LOAD GFRZ WITH A NEW FUNCTION CALL TO FAC :
      CORE MADDR,4089: 160006B,120000B;
*ENTRY IN FUNCTION TABLE:
      CORE MADDR,224: 120000B,2B,1025B,0B;
*THE LINE TABLE IS :
      CORE MADDR,532: 4,0,650,660,670,674;
*
*FAC 3
      CORE MADDR,679:001140B,000000B,114000B,122004B;
*FAC7
      CORE MADDR,683:114001B,122004B;
*FAC 7
      CORE MADDR,685:001560B,000000B,114000B,122004B;
*FAC 1
      CORE MADDR,689:000500B,000000B,114000B,122004B;
*****
*THE GLOBAL PHANTOM TABLE :
* 0      = 114000 = FAC
* 1      = 114001 = FAC7
*****

END
$

```

Figure 4.12 - The factorial: Translator output

```

END AT DOWNFUZZ+2
ORF PP,679;
DSB ERROR,FCNCALL,FRETURN,PRIN1AR,STARTLINE;
GOS
SBCALCMODE;
GOTEST;
CALCMODE
PR;
FCNCALL
PR;
STARTLINE
PR;
STARTLINE
ORO FREE;
FREE(27): 2
PR;
FCNCALL
OCE MADDR,7460B,7510B;
MADDR,7460: 0
MADDR,7461: 0
MADDR,7462: 0
MADDR,7463: 0
MADDR,7464: 0
MADDR,7465: 0
MADDR,7466: 0
MADDR,7467: 0
MADDR,7470: 140002
MADDR,7471: 1252
MADDR,7472: 0
MADDR,7473: 7500
MADDR,7474: 4
MADDR,7475: 1025
MADDR,7476: 120000
MADDR,7477: 6
MADDR,7500: 170000 } Z is undefined
MADDR,7501: 0
MADDR,7502: 1140 } N = 3
MADDR,7503: 0
MADDR,7504: 140000
MADDR,7505: 0
MADDR,7506: 0
MADDR,7507: 0
MADDR,7510: 0
PR;
STARTLINE

```

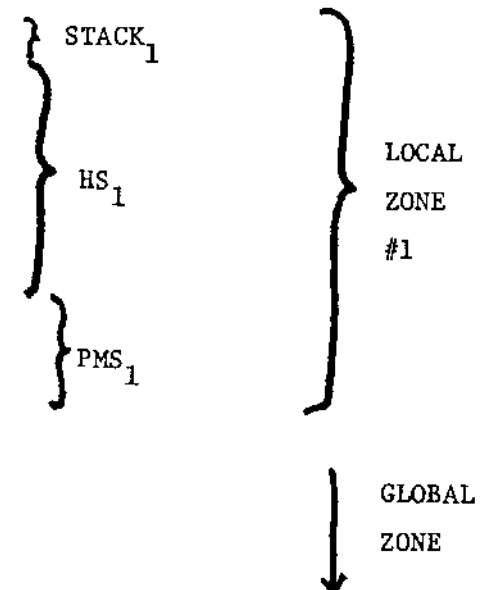


Figure 4.13 - The first call is encountered

CE MADDR, 7440B, 7510B;

MADDR, 7440: 0

MADDR, 7441: 0

MADDR, 7442: 0

MADDR, 7443: 0

MADDR, 7444: 0

MADDR, 7445: 0

MADDR, 7446: 0

MADDR, 7447: 0

MADDR, 7450: 0

MADDR, 7451: 0

MADDR, 7452: 0

MADDR, 7453: 0

MADDR, 7454: 0

MADDR, 7455: 1231

MADDR, 7456: 7502

MADDR, 7457: 7464

MADDR, 7460: 4

MADDR, 7461: 1025

MADDR, 7462: 120000

MADDR, 7463: 6

MADDR, 7464: 170000 } UV

MADDR, 7465: 120001

MADDR, 7466: 1100 } 2

MADDR, 7467: 0

MADDR, 7470: 140002 LN

MADDR, 7471: 1252

MADDR, 7472: 0

MADDR, 7473: 7500

MADDR, 7474: 4

MADDR, 7475: 1025

MADDR, 7476: 120000

MADDR, 7477: 6

MADDR, 7500: 170000 } UV

MADDR, 7501: 0

MADDR, 7502: 1140 } 3

MADDR, 7503: 0

MADDR, 7504: 140000 LN

MADDR, 7505: 0

MADDR, 7506: 0

MADDR, 7507: 0

MADDR, 7510: 0

0

HS<sub>2</sub>

PMS

STACK<sub>1</sub>

HS<sub>1</sub>

PMS

STACK

Figure 4.14 - The second call is generated

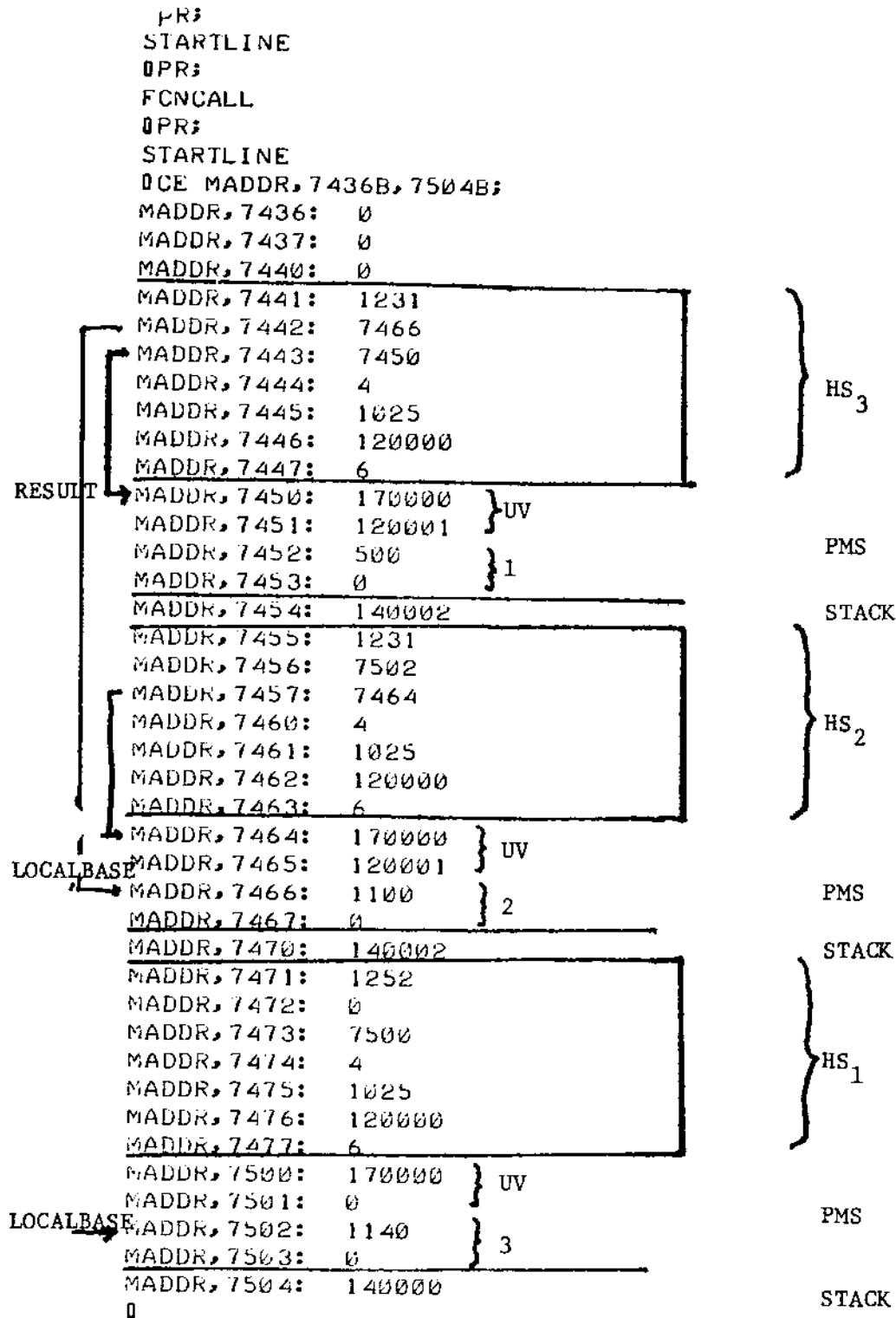


Figure 4.15 - Third Call

```

PR;
STARTLINE
QPR;
FCNCALL
QROAR0,AR1;
AR0(24): 0
AR1(25): 0
QPR;
STARTLINE
QCEMADDR,7416B,7504B;
MADDR,7416: 0
MADDR,7417: 0
MADDR,7420: 0
MADDR,7421: 0
MADDR,7422: 0
MADDR,7423: 0
MADDR,7424: 0

```

Figure 4.16 - Fourth and Final Call

```

MADDR,7425: 1231
MADDR,7426: 7452
MADDR,7427: 7434
MADDR,7430: 4
MADDR,7431: 1025
MADDR,7432: 120000
MADDR,7433: 6

```

HS<sub>4</sub>

```

MADDR,7434: 170000
MADDR,7435: 120001
MADDR,7436: 0
MADDR,7437: 0 } 0

```

```

MADDR,7440: 140002

```

```

MADDR,7441: 1231
MADDR,7442: 7466
MADDR,7443: 7450
MADDR,7444: 4
MADDR,7445: 1025
MADDR,7446: 120000
MADDR,7447: 6

```

HS<sub>3</sub>

```

MADDR,7450: 170000
MADDR,7451: 120001
MADDR,7452: 500
MADDR,7453: 0

```

```

MADDR,7454: 140002

```

```

MADDR,7455: 1231
MADDR,7456: 7502
MADDR,7457: 7464
MADDR,7460: 4
MADDR,7461: 1025
MADDR,7462: 120000
MADDR,7463: 6

```

HS<sub>2</sub>

```

MADDR,7464: 170000
MADDR,7465: 120001
MADDR,7466: 1100
MADDR,7467: 0

```

```

MADDR,7470: 140002

```

```

MADDR,7471: 1252
MADDR,7472: 0
MADDR,7473: 7500
MADDR,7474: 4
MADDR,7475: 1025
MADDR,7476: 120000
MADDR,7477: 6

```

HS<sub>1</sub>

```

MADDR,7500: 170000
MADDR,7501: 0
MADDR,7502: 1140
MADDR,7503: 0

```

```

MADDR,7504: 140000

```

Figure 4.17 - The returns

```

PR;
SVLOOP+22B
OPR;
S
0/
STARTLINE:  VMD ← FREE OR LNCODE;
OPR;
FRETURN
DOE MADDR,7434B,7435 +B;
MADDR,7434:  500
MADDR,7435:  0
OPR;
STARTLINE
DOE MADDR,743+50B,7451B;
MADDR,7450:  500
MADDR,7451:  0
} result, level 3
DO STACK;
STACK(35):  7441
OPR;
FRETURN
OPR;
STARTLINE
DOE MADDR,7464B,7465B;
MADDR,7464:  1100
MADDR,7465:  0
} result, level 2
OPR;
FRETURN
OPR;
STARTLINE
DOE MADDR75+1,7500B,7501B;
MADDR,7500:  1540
MADDR,7501:  0
} result, level 1
OPR;
FRETURN
OPR;
PRINTAR
DO AR0,AR1;
AR0(24):  1540
AR1(25):  0
} final result, level 0
0

```



Figure 4.19 - Factorial 1

```

    PR;
    CALC MODE
    @PR;
    FCNCALL
    @PR;
    FCNCALL
    @PR;
    FRETURN
    @PR;
    FRETURN
    @PR;
    PRINTAR
    @RO AR0,AR1;
    AR0(24): 500
    AR1(25): 0
0

```

} 1



# **CHAPTER    5**

## **THE OPERATORS**

1.    APL Operators
2.    A Front End Processor for Scalar Operators
3.    Other Operators
4.    Indexing as a Dyadic Operator

## 1. General

The semantic actions taken by the Parser upon detection of an operator have already been presented in section C3 of Chapter IV. This chapter considers exclusively the case of APL's numeric operators (scalar or mixed), already identified as dyadic or monadic.

It should be first noted that not all APL operators can or need reside within the control storage and indeed, it is sufficient to implement a basic spanning set of operators within the microcode, all others being specified as software APL functions utilizing this basic set. In view of the restricted size of the control storage, only the operators deemed to be most frequently used have been microcoded, all others being software functions (see appendix for a complete list and appendix G for listings of the software operators).

## 2. Scalar Operators: An Operator Front End

Scalar operators in APL are the only ones which operate in a fairly uniform manner on scalars and arrays. All other non-standard operators have been grouped under the denomination of mixed operators. Scalar operators are therefore handled by a common Front End processor in charge of the sequencing combinations for the various types of arguments. Only the procedure for dyadic operators need be described in detail since monadic operators are handled as a special case of the dyadic routines.

The two arguments of a dyadic scalar operator are therefore tested for type and a branch to the appropriate routine made: SCASCA for scalar-scalar, SCAVEC for scalar-array, VECSCA for array-scalar,

VECVEC for array-array, NULL for any null argument.

1 - The SCASCA routine simply constitutes a branch through an operator jump table to the appropriate operator routine.

2 - VECVEC runs conformability checks on the two array operands then allows for two cases, according to the type of the arrays (scalar/character). In each case, it initially obtains a block for the result (possibly re-using one of the array-operands) and enters an iteration loop where it loads the left element into AL, the right argument into AR, calls the OPERATE jump table, and returns to the SVLOOP which updates the result pointer, stores the computed value and reenters the loop if the counter is not null.

3 - VECSCA also uses two different routines according to the array type, fetching the left element, loading it in AL, then calling OPERATE and using SVLOOP to store the result and test for loop termination, before reloading AR with the fixed scalar and returning to fetch the next left element.

4 - SCAVEC simply exchanges the AR and AL (right and left argument) and branches to VECSCA.

All operators return upon completion of the sequence to an OPRETURN routine in charge of clean-up operations. The result is placed in the AR (scalar, AP, Null) and if any array were used, they are tested for releasability and explicitly released if allowed. A dyadic operator exits in state 1 and resumes parsing (PARSENEXT). A monadic operator could exit in state 2 and resume parsing. In practice, since a look-ahead is needed to identify the operator as monadic, the next operator has already been fetched from core

and has been placed in a fast register, in order to avoid a needless core fetch upon OPRETURN. Instead of going to PARSENEXT, the next operator which had been saved is therefore installed in the appropriate register and a branch is made to OPERATOR, equivalent to parsing of an operator. The listings for the microcoded scalar operators are given in appendix M. It can be noted that the floating point routines (+ - × ÷) are sizeable and relatively slow. A significant improvement in the efficiency of the interpreter could therefore be gained by the utilization of hardware floating point units which would provide added speed in the execution of these operators and room for the implementation in microcode of other operators.

### 3. Other Operators

Mixed operators cannot have the benefit of a common front-end and must be implemented on an individual basis. They appear in Appendices G and M. An exceptional dyadic operator is the assignment and it is now presented in detail.

Upon detection of the ←, the next token is examined for a phantom or a ]. If a ] is found, the OPR and the AR are deferred as in the case of any other dyadic operator, and parsing resumes in state 0.

If a phantom is found, it is verified not to be a function call and then examined further. If a PM or AP, the pointer to its value is obtained from the PMS entry. If a scalar, or Null, the pointer is the PMS address. If the new value is a non-releasable array, a copy is created.

The previous value, if array, is then released if releasable and the new value, or its pointer if an array, is entered within the PMS.

The next token is then examined; if LD, it was a "simple" assignment and the LD routine previously outlined is entered. If not LD, a middle of line assignment has been performed. The AR contains the value or a pointer to it and execution resumes with decyphering of the token.

#### 4. Multidimensional Indexing as a Dyadic Operator

APL indexing is defined in terms of its syntactic and semantic properties and shown to be semantically equivalent to a new operator, " $\alpha$ ", whose syntax characterizes indexing as a special dyadic operator.

A formal algorithm is developed using a new symbolism, the "odometer", for the simple realization of the  $\alpha$  (indexing) operator and the essential features of the practical implementation are presented.

##### A. Multidimensional Indexing

APL's indexing operation is syntactically defined by the following BNF:

$$\begin{aligned} \langle \text{INDEXED EXPRESSION} \rangle &::= \langle \text{EXPRESSION} \rangle [\langle \text{INDEXING LIST} \rangle] \\ \langle \text{INDEXING LIST} \rangle &::= \{ \langle \text{EXPRESSION} \rangle \} \{ ; \langle \text{INDEXING LIST} \rangle \} \end{aligned}$$

where  $\{ \dots \}$  denotes 0 or 1 times  $\dots$ . (The syntax for  $\langle \text{EXPRESSION} \rangle$  may be found in Appendix B, Group 4.)

Consider  $R \leftarrow T[P;Q]$  where P and Q are respectively 2- and 3-dimensional arrays. Semantically, APL's indexing is defined by

$$R[I;J;K;L;M] \equiv T[P[I;J];Q[K;L;M]]$$

$$\text{with } \begin{cases} \rho R = (\rho P), \rho Q & = N1 \ N2 \ N3 \ N4 \ N5 \\ \rho\rho R = (\rho\rho P) + \rho\rho Q & = 5 \end{cases}$$

where I, J, K, L, M are integer indices with  $I \in [1, N1]$ , etc.

When the indexing list is an APL vector, or

$$\begin{aligned} &R \leftarrow T[P] \quad \text{with } \rho\rho P = 1, \\ \text{then } &\begin{cases} \rho R = \rho P \\ \rho\rho R = \rho\rho P = 1 \end{cases} . \end{aligned}$$

Indexing may be applied to any array of rank  $> 1$ . Indices, however, are not restricted as to rank, and by convention, even "Null" may be used as an index to represent

1 <number of elements for this dimension>

or even all elements, when enclosed in indexing brackets. The following are explicit examples.

Example 1: matrix indexing a vector

A ← 'OAUHYRWE'

B ← 3 3 ρ 4 1 7 2 6 8 5 1 3

OAUHYRWE

B

4 1 7

2 6 8

5 1 3

A[B]

HOW

ARE

YOU

Example 2: use of Null index

A ← 2 3 ρ 2 6

A

2 6 2

6 2 6

A[1;]

2 6 2

A[;2]

6 2

A[;1]

2 6

Example 3: element retrieval

(2 3 ρ 16)[1;2]

2

#### B. A Dyadic Indexing Operator

A dyadic index operator  $\alpha$  is defined as follows:

(1) <INDEXED EXPRESSION> ::= <EXPRESSION> $\alpha$ < $\alpha$ LIST>

(2)  $\langle \alpha \text{LIST} \rangle ::= (\langle \text{INDEXING LIST} \rangle)$

with  $\langle \text{INDEXING LIST} \rangle$  defined as before.

The parentheses perform the usual grouping function and dyadic  $\alpha$  performs the actual indexing operation described in the preceding section.

This representation reveals the significance of APL's indexing: first the index  $\alpha$  is a true APL dyadic operator. Second,  $\alpha \text{LIST}$ , the right argument, is a data-type special to this operator and therefore syntactically inconsistent with the remainder of the language.

APL normally allows two basic data types: numeric (including logical) and character, and one structure: arrays. APL arrays are restricted to be homogeneous, i.e., composed exclusively of alphanumerics, scalars or logicals. This structure lacks generality since arrays may not have other arrays as elements and restricts seriously the practicality of the language in a structure manipulation environment. It is therefore interesting to note how this restriction has been *half-way* relaxed in APL\360 for the index operator. Assuming that no special data-type is introduced for exclusive use in indexing, the index operator could be defined as

(3)  $\langle \text{INDEXED EXPRESSION} \rangle ::= \langle \text{EXPRESSION} \rangle \alpha \langle \text{EXPRESSION} \rangle$  .

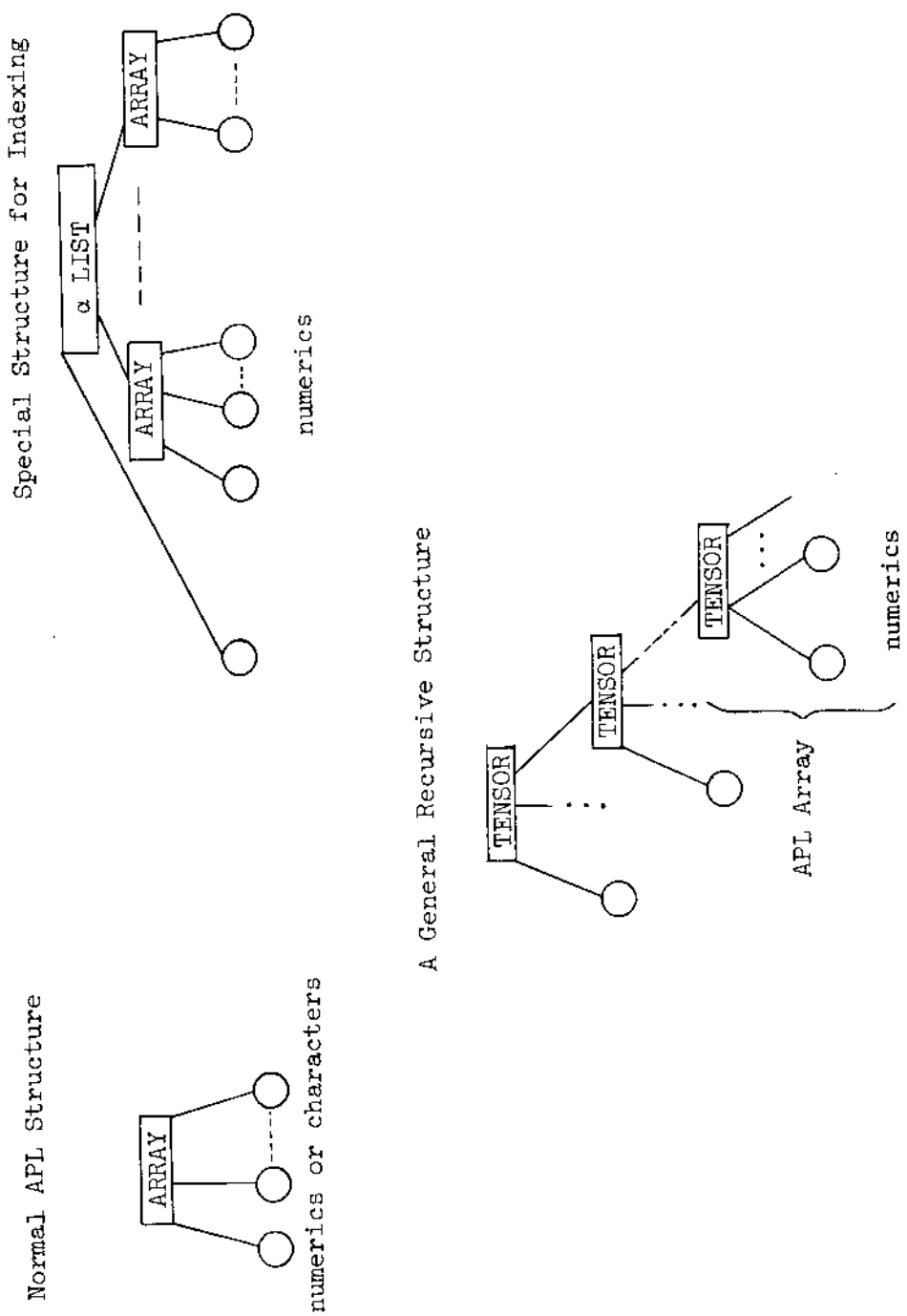
Its pure dyadic nature is apparent. This case covers all "normal" indexing operations in the usual sense and selects an element within a multi-dimensional array:

```
if      V ← 2 2 ρ 2 4
then    V α 2 2   selects V[2;2] = 4 .
```

However, as indicated above, APL provides an extended data-type for exclusive use in indexing: the right  $\langle \text{EXPRESSION} \rangle$  of (3) is allowed to be an  $\langle \alpha \text{LIST} \rangle$ , i.e., either an array of scalars or a list of (numeric) arrays. This  $\alpha \text{LIST}$ , however, is only a half-step towards a general data-type since the arrays



Figure 5.1 - Data Structures for APL



which may appear within the list are still restricted to be exclusively composed of numerics and not other arrays: there is no recursion possible within this structure other than by explicitly nesting  $\alpha$  operators within the  $\alpha$ LIST.

This special data-type appears nowhere else in the language and needs therefore to be appropriately identified: it is a collection of arrays or numerics so that two symbols are needed: a grouping symbol and separators. As can be seen in (2) above, parentheses can perform the grouping function. Blanks, however, cannot perform the separating function as in literal arrays for a distinct separator is needed between an array and the adjacent element if the resulting structure is not to be just another array. Separators used in APL are semi-colons. Note that if a special symbol were used as separator exclusively within an  $\alpha$ LIST, say  $\textcircled{;}$ , it would not be necessary to use a grouping symbol, but that for consistency with APL's right to left order of evaluation, indexing should be written as  $\langle\alpha\text{LIST}\rangle\alpha\langle\text{EXPRESSION}\rangle$ .

It has now been shown that APL indexing should be regarded as a dyadic operator which requires a special data-type, the  $\alpha$ LIST. A special technique is now introduced for this data type, based on its index order representation.

### C. Index Order Representation

APL arrays are most naturally represented in *index order* or row-major order and an understanding of the representation is essential to the understanding of the algorithm presented in the next section.

An example should make it clear:

```

E ← 2 4 3 0 124
E
1 2 3
4 5 6
7 8 9
10 11 12

```

```

13 14 15
16 17 18
19 20 21
22 23 24 .

```

Let us ravel E to obtain its index representation:

```

,E
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

```

#### D. Element Location

A right to left iterative computation for the location of each element in the row major order representation is given in figure below. The access equation is shown to be:

$$\text{address of } V[E_1; E_2; \dots; E_n] = \text{Vbase} + \sum_{i=1}^{n-1} ((E_i - 1) \times \prod_{j=i+1}^n D_j) + E_n$$

in base 1 indexing.  $D_i$  is the number of elements for the  $i^{\text{th}}$  dimension.

In particular, this access formula can be verified for the extremal case:

Example:  $V[D_1; D_2; \dots; D_n]$

Applying the above formula:

address of $V[D_1; \dots; D_n] = \text{base} +$	or:
$(D_1 - 1) \times D_2 \times \dots \times D_n$	$\prod_{j=1}^n D_j - \prod_{j=2}^n D_j$
$+ (D_2 - 1) \times D_3 \times \dots \times D_n$	$+ \prod_{j=2}^n D_j - \prod_{j=3}^n D_j$
$+ \dots$	$+ \dots$
$+ (D_{i-1} - 1) \times D_i \times \dots \times D_n$	$+ \prod_{j=i-1}^n D_j - \prod_{j=i}^n D_j$
$+ (D_i - 1) \times D_{i+1} \times \dots \times D_n$	$+ \prod_{j=i}^n D_j - \prod_{j=i+1}^n D_j$
$+ \dots$	$+ \dots$
$+ (D_{n-1}) \times D_n$	$+ \prod_{j=n-1}^n D_j - D_n$
$+ D_n$	$+ D_n$
	$= \prod_{j=1}^n D_j$

which is obviously verified on the graphical representation of figure .



### E. The Odometer

The algorithm has been invented independently and can be compared to a similar mechanism recently published in Hassitt [1972].

#### E1. Array subscripts

Index representation allows, at the cost of one multiplication per index (after the first one), efficient retrieval of elements within an n-dimensional vector indexed by scalars. The location A of the element is:

$$A = B + \sum_{i>1} (X_i - 1) \times D_{i-1} + X_1 = \text{address of } M[X_1;X_2;X_3;\dots;X_n] \quad (1)$$

where B is the base of the vector,  $X_i$  the index for dimension i, and  $D_i$  the number of elements along dimension i.

Let us now show how the method can be extended to the case of an n-dimensional vector indexed by arrays.

Recall that  $M [ ]$  with  $\rho M = 2 \ 3$  is represented in row major order as:

$M[1;1], M[1;2], M[1;3], M[2;1], M[2;2], M[2;3]$ .

In the general case where  $R \leftarrow T[P;Q]$ , (with P and Q 2-dimensional),  $R [ ]$  is:

$R[1;1;1;1], R[1;1;1;2], R[1;1;1;3], \text{ etc.},$

or:

$T[P[1;1]; Q[1;1]], T[P[1;1]; Q[1;2]], T[P[1;1]; Q[1;3]], \text{ etc.}$

## E2. The "Odometer" Symbolism

All 4 indices of  $R [ i; j; k; l ]$  are displayed in a window, the "odometer":

i	j	k	l
---	---	---	---

Figure 5.3 - The Odometer

Now, let the 4 wheels cycle so as to generate all the elements of  $R$  in row major order and watch them go by in the window. We are now watching the "indexing odometer". The analogy should be obvious:

- $l$  will cycle from 1 to  $D_4$ . The transition from  $D_4$  back to 1 will trigger  $k$ .  $l$  keeps cycling.

- $k$  is incremented by 1 every time  $l$  goes from  $D_4$  to 1. It will thus cycle  $D_4$  times slower than  $l$  from 1 to  $D_3$ . The transition from  $D_3$  back to 1 triggers  $j$ .

- $j$  is incremented by 1 every time  $k$  goes from  $D_3$  to 1. The transition from  $D_2$  to 1 triggers  $i$ .

- $i$  is incremented by 1 up to  $D_1$ . Once  $i$  reaches the value  $D_1$ , the odometer is:

$D_1$	$D_2$	$D_3$	$D_4$
-------	-------	-------	-------

Figure 5.4 - Final Odometer

and our trip is completed: all the elements of  $R$  have been generated.

Conversely, any four-dimensional vector  $R$  represented in row major order has a four-wheel "indexing odometer" associated with the representation.

In our example,  $R \leftarrow T[P;Q]$ ,  $T$  has a two-wheel odometer, one per index. Each index in turn, being a matrix in row major order, has a two-wheel odometer:

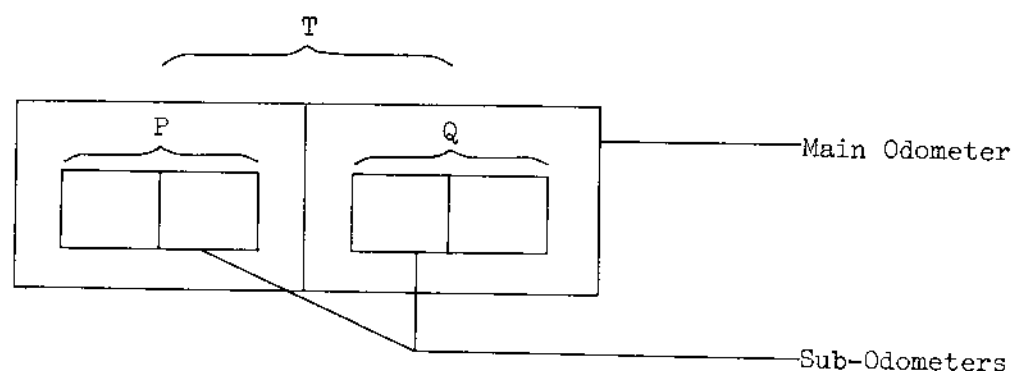


Figure 5.5 - The Two-Wheel Odometer

This results in  $R$  having finally a four-wheel odometer.

This symbolism gives us the key to a simple indexing algorithm: since the odometer characterization of any vector is in one to one correspondence with its row major order representation, there is no need to carry sub-odometers; they are implicitly carried by the representation. In other words, cycling the sub-odometer for  $Q$  is equivalent to moving a pointer from left to right along its row major order representation.

Thus, cycling the odometer for  $T$  can be reduced to the cycling of its main odometer, with one wheel per index.



### E3. The "Ripple" Mechanism

The odometer permits general indexing, as defined in APL. In conventional indexing, where a single element is accessed, the odometer remains fixed, pointing to the element. In multidimensional indexing, where one or more of the indices are arrays, the odometer will cycle. The cycling will generate a sequence of odometer frames. Each odometer frame points to one element of the result. Once the odometer will have cycled, all elements of the result will have been generated in row-major order.

The "mechanical" operation of the odometer involves cycling its rightmost wheel, then the one to its left, and so on. Each wheel which has completed a cycle generates a "ripple" to the left. The ripple activates in turn the rotation of a new wheel. This mechanism is now described.

The digit  $n$  appearing in a window now corresponds directly to the  $n^{\text{th}}$  element of that dimension and we only have one pointer associated with each index, even though the index may in turn be of arbitrary rank.

The indexing algorithm is now very simple. One pointer (or "digit") is carried per index and initialized to 1 (the first element of that dimension) (null is treated as the iota of the number of elements for the dimension). The last pointer is initialized to 0. The "ripple" is now sent to the rightmost frame of the odometer (the pointer for the last index) causing it to advance one notch. If this index were a scalar (one element), it would simply reset itself and propagate the ripple to the left. More generally, every time a digit cycles back from  $D_i$  to 1, the ripple is propagated.

The odometer is allowed to settle and the resulting digits  $X_i$  are used in (1) to access the vector element which becomes the next element of the result.

If the ripple has overflowed left of the odometer, we are done. Otherwise, it is gated again to the rightmost digit and the process cycles until completion.

The mechanism is completely general and will retrieve one element of a vector, as well as select some of its dimensions or create a higher order vector through the use of multi-dimensional indices.

An Example

The following matrix is considered:

$$F \leftarrow 3 \ 4 \ \rho \ 13$$

F

1	2	3	1
2	3	1	2
3	1	2	3

F[F;]

1	2	3	1
2	3	1	2
3	1	2	3
1	2	3	1

2	3	1	2
3	1	2	3
1	2	3	1
2	3	1	2

3	1	2	3
1	2	3	1
2	3	1	2
3	1	2	3

F[:,F]

1	2	3	1
2	3	1	2
3	1	2	3

2	3	1	2
3	1	2	3
1	2	3	1

3	1	2	3
1	2	3	1
2	3	1	2

The odometer technique is now applied to generate  $F[:,F]$ .  $F$ 's row-major order representation is:

$,F$   
1 2 3 1 2 3 1 2 3 1 2 3 .

Its odometer is a two-wheel odometer:

i	j
---	---

where  $j$  cycles from 1 to 12 ( $F$ 's number of elements)

and  $i$  cycles from 1 to 3 ( $F$ 's number of rows).

The resulting dimensions are: 3 3 4. The resulting array's row-major order representation is generated as follows:

1	1 to 12
---	---------

⇒
1 2 3 1 2 3 1 2 3 1 2 3

generated from  $F$ 's first row: 1 2 3 1

by selecting from it the  $k^{\text{th}}$  element where  $k$  is the  $j^{\text{th}}$  element of  $,F$  and  $j$  cycles from 1 to 12. Similarly, letting the ripple propagate:

2	1 to 12
---	---------

⇒
2 3 1 2 3 1 2 3 1 2 3 1

generated from  $F$ 's second row: 2 3 1 2

and:

3	1 to 12
---	---------

⇒
3 1 2 3 1 2 3 1 2 3 1 2

generated from  $F$ 's third row: 3 1 2 3

and the ripple overflows, terminating the algorithm.

The complete index representation of  $F[:,F]$  is therefore:

1 2 3 1 2 3 1 2 3 1 2 3 2 3 1 2 3 1 2 3 1 2 3 1 3 1 2 3 1 2 3 1 2 3 1 2

and the result (dimensions 3 3 4) is:

```

1 2 3 1
2 3 1 2
3 1 2 3

2 3 1 2
3 1 2 3
1 2 3 1

3 1 2 3
1 2 3 1
2 3 1 2 .

```

The following example is left as an exercise to the reader:

```

      F[F[F;];]
1 2 3 1
2 3 1 2
3 1 2 3
1 2 3 1

2 3 1 2
3 1 2 3
1 2 3 1
2 3 1 2

3 1 2 3
1 2 3 1
2 3 1 2
3 1 2 3

1 2 3 1
2 3 1 2
3 1 2 3
1 2 3 1

2 3 1 2
3 1 2 3
1 2 3 1
2 3 1 2

3 1 2 3
1 2 3 1
2 3 1 2
3 1 2 3

1 2 3 1
2 3 1 2
3 1 2 3
1 2 3 1

2 3 1 2
3 1 2 3
1 2 3 1
2 3 1 2

```

[illegible]

## F. Implementation

The merits of an iterative address computation are now examined. In the case of conventional indexing, where all  $E_i$ 's are positive integers, an iterative computation does not offer any speed advantage.

However, in the case of generalized APL indexing, where some or all of the  $E_i$ 's may be multidimensional arrays, the efficiency of an algorithm depends critically on the number of multiplications to be performed. Typically a multiply operation takes 5 to 50 times as long to complete as a core memory access, depending on the implementation (hardware, firmware, software) and word length. It becomes therefore imperative not to perform unnecessary multiplications. The odometer symbolism allows a simple method to display the solution. Only one wheel of the odometer is cycling at any time. All other windows remain fixed. An optimum algorithm will therefore recompute only those partial products which have been changed, i.e., will affect only the wheel being cycled. Partial products for all other windows remain unchanged.

An iterative computation will permit to achieve this optimum efficiency.

In order to permit an iterative computation, the equation developed in section D for element access is now rewritten as follows:

$$\text{Define} \quad \text{INC}(i) = (E_i - 1) \times \prod_{j=i+1}^n D_j, \quad i = 1, \dots, n-1$$

$$\text{with} \quad \text{INC}(n) = E_n.$$

Further, let 
$$\text{PROD}(i) = \prod_{j=i+1}^n D_j$$

with 
$$\text{PROD}(n) = 1$$

We then have: 
$$\text{PROD}(i-1) = D_i \times \text{PROD}(i) \quad (1)$$

and 
$$\text{INC}(i) = (E_i - 1) \times \text{PROD}(i) \quad (2)$$

and the access equation developed in section D becomes:

$$\text{address of } V[E_1; E_2; \dots; E_n] = \text{Vbase} + \sum_{i=1}^n \text{INC}(i)$$

where  $\text{INC}(i)$  and  $\text{PROD}(i)$  are computed through the iteration formulas (1) and (2) above.

Computing the address of  $V[E_1; E_2; \dots; E_n]$  iteratively from high to low indices,  $\text{PROD}(i)$  is computed using a single multiplication,  $\text{INC}(i)$  using one subtraction and one multiplication.

In order to perform efficient element retrieval, it is therefore necessary to store at least 5 quantities per "index" (abscissa).

- The  $\text{PROD}$  and  $\text{INC}$  above necessary to the computation
- The current and final odometer
- The indexing element, or a pointer to it

These five entries can be conceptualized as constituting the odometer block, illustrated below (figure ). It has been implemented here as a linear ordering of the 6 word entries per abscissa in the Stack. At the time an  $E_i$  is parsed, it is pushed into the Stack along with 4 empty words. Upon activation of the indexing operator, this 6-word entry becomes the subblock of the Odometer for absciss  $i$ .



The actual implementation of the indexing algorithm is now straightforward. A left to right pass is made over the odometer blocks in order to verify that the number of  $E_i$ 's is equal to the dimensionality of the array  $V$  being indexed. The initial right-to-left pass (RLPASS) then loads all initial values within the odometer blocks. In the case where all  $E_i$ 's are integers, indexing then exits to retrieve the element of  $V$  selected. In the case of generalized indexing, the RIPPLETHRU routine is then called on repetitively until the odometer reaches its final value. RIPPLETHRU cycles everytime the proper odometer window by one notch, recomputes the INC and accesses the  $V$ -element selected, which becomes part of the row-major order representation of the resulting array.

Figure 5.6 - The Odometer Blocks

ABSCISSA	1	i	n-1	n	
INC	$(E_1^{-1}) \times \text{PROD}_1$	$(E_i^{-1}) \times \text{PROD}_i$	$(E_{n-1}^{-1}) \times \text{PROD}_{n-1}$	$E_n$	current Odometer
DIGIT	$d_1$	$d_i$	$d_{n-1}$	$d_n$	
PROD	$D_2 \times D_3 \times \dots \times D_n$	$D_{i+1} \times \dots \times D_n$	$D_n$	1	final Odometer
VDIM	$D_1$	$D_i$	$D_{n-1}$	$D_n$	
ELEMENT	$E_1$	$E_i$	$E_{n-1}$	$E_n$	

## **CHAPTER 6**

### **PERSPECTIVE**

1. General
2. The Evolution of Microprogramming
3. A System Architecture
4. A Language Machine
5. Other Processors
6. Performance
7. Conclusions

## 1. General

The feasibility and the methods for implementing a dedicated APL processor have been demonstrated in the preceding chapters. It is not intended here to investigate the vast problem of operating system design and the discussion is limited to a system architecture for APL time-sharing. A global architecture for real-time APL service is presented in section 3. The APL machine for which the prototype interpreter has been coded is then introduced in section 4 and its performance is surveyed in section 5. Other processors for APL execution are examined in section 6. The summary and concluding remarks appear in section 7. In order to preface the hardware concepts to be discussed, the evolution of microprogramming is examined first.

## 2. The Evolution of Microprogramming

### A. New Directions in Microprogramming

From the introduction of the concept of microprogramming twenty years ago to present days, microprocessors have been gradually used to perform a variety of new functions.

Very recently, however, microprocessors are encountering an unprecedented success in many novel areas of computing hardware utilization. We will present here a brief survey of traditional uses, introduce a conceptual model for microprocessors, analyze some of the causes of this new and rapid development and present new directions for the development of the design and utilization of contemporary microprocessors.

### B. Evolution of Microprogramming

Microprogramming was introduced by Wilkes in 1951 and represented a systematic approach to the design of a coherent and simple control unit on a conventional processor. In 1964, with the IBM 360 series, the concept was applied to the design of compatible instruction sets for machines of differing speed, size and architecture. In 1965 and 1966, new applications were developed: controlling of devices and emulation of other computers by microprocessors. At about the same time, several papers appeared pointing to the theoretical desirability of implementing directly a high level language interpreter within control storage. In 1969, microprogramming was used extensively within the 370 series, using writable control storage, providing compatibility of instruction sets, emulation of the previous series, very refined diagnostic capabilities and an expansion capability for additional features or future modifications.

### C. A Microprocessor Model

There is no attempt here to redefine microprogramming, but simply to present a simple conceptual model for a contemporary microprocessor. Such a processor is able to transfer information between several memory levels. It is equipped with an arithmetic, Boolean and shift capability during these transfers, and with the capability to initiate the various transfers asynchronously.

It references five memory levels communicating between two or three buses via an arithmetic-shift unit (see figure 6.1). Note that the presence of the control storage box among the logical memory levels implies an access width compatible with the bus width, i.e. a scheme of vertical microprogramming, where the width of a control storage word is compatible with the bus transfer capability and the word length of other memory levels. Vertical microprogramming is different from the horizontal microprogramming approach, where each bit of the control word directly controls a gate. In vertical microprogramming bits are grouped into fields and are interpreted in the context of the instruction type: there are a few levels of hardware interpretation which reduce the potential parallelism of individual operations specified within a micro-instruction, but provide a simple and clean instruction set as well as a short control word which can use the same busing facilities as other words within the system.

Typical instructions are of the type:

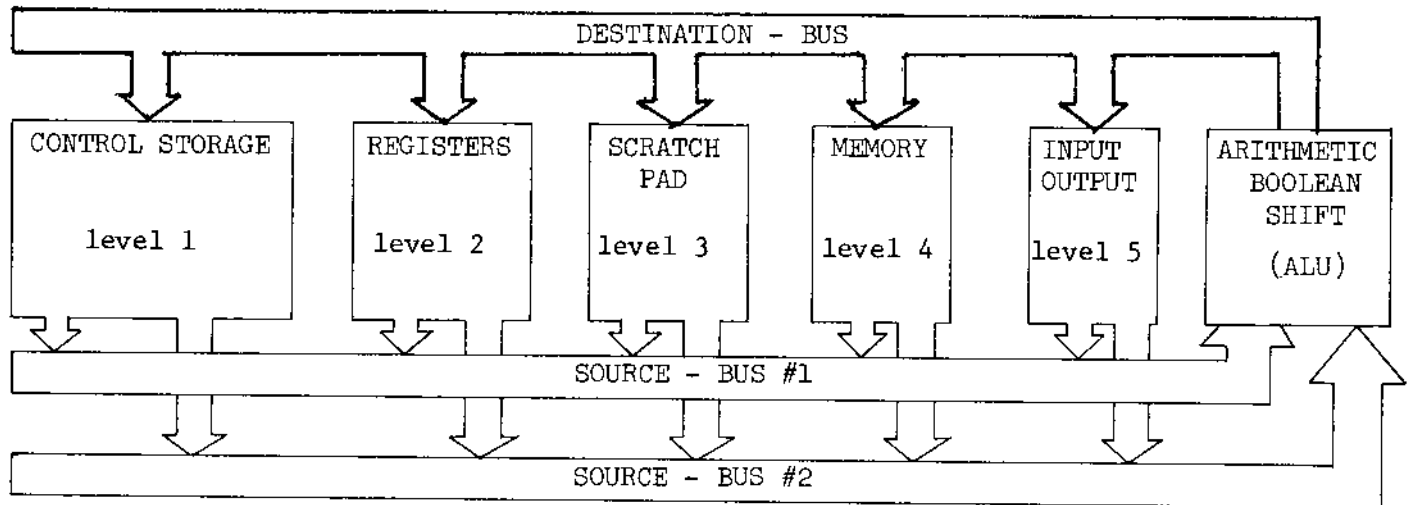
• <source level i> <arithmetic operator> <source level j>

<shift operation and special modifiers> → <destination level k>

· goto <location in control storage> if <bit> in <register> is zero (one).

Naturally, on some simple machines, level j may be implicit (accumulator) and certain access restrictions may hold for transfers between levels i, j and k (intermediate transfers to special registers).

Figure 6.1 - Conceptual Model of a Microprocessor





Note that this organization is logical rather than physical, and there is a possibility for some of these levels not to be implemented or to reside on the same physical medium. Thus, in the new IBM 370 series, control and conventional storage reside within the same physical device (IC memory), but are logically distinct. A return to the old days also appears as a possibility (Tucker and Flynn, 1971) with control storage and registers residing within a common IC storage device.

However, such approaches are essentially motivated by economic considerations and tend to decrease the flexibility of hierarchical and asynchronous memory levels.

One essential feature of this representation is the fact that not only control storage, registers and possibly scratch-pad appear as memory levels to the microprocessor, but that memory proper and IO do as well. In the case of the memory box, and possibly any other box, it should be noted that the data deposited into the box are not necessarily the ones that will be retrieved from it: there may be communication with the outside world through a memory box which is being accessed through several ports.

Similarly, *the IO box* is viewed, not as a static memory box, but as two semi-infinite memories, one for output from the destination bus and one for input to the source buses. Both may be accessible through a common logical slot, but use separate physical connections.

The IO box may represent a single IO controller, or several IO devices connected independently and in parallel between the buses.

*The registers* consist of 4 or more high-speed general-purpose registers with a few dedicated registers holding special flags or equipped with special connections or with local special-purpose elementary hardware.

*The scratch-pad* is meant to represent the highest (fastest) level of conventional storage addressed by means of an explicit address placed in a separate register as opposed to the addressing scheme which is used for registers and is typically used in vertical microprogramming.

*The memory box* represents a conventional level of mass storage, but could consist of several levels of differing capacities and speeds, suspended between the source and destination buses.

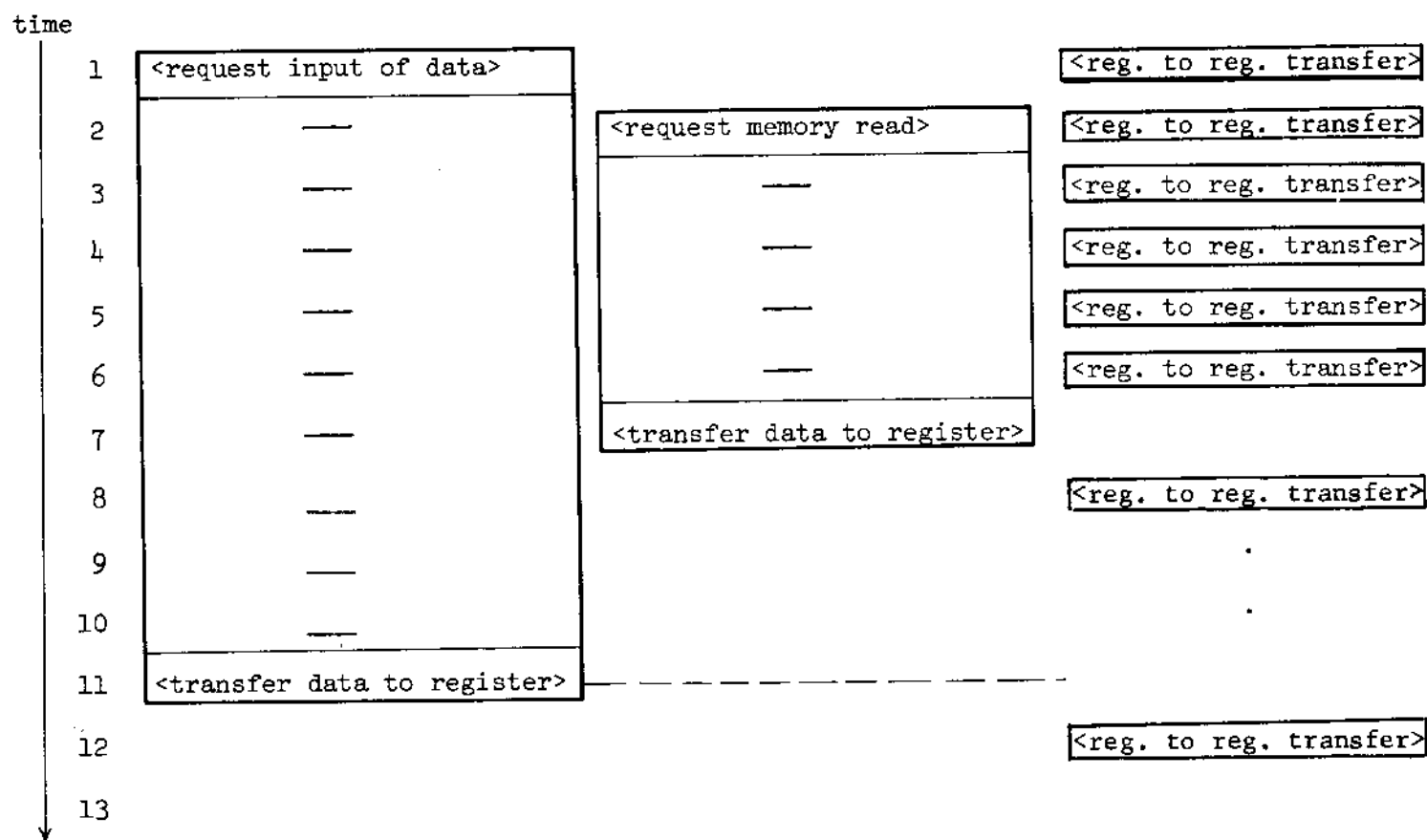
*The arithmetic, Boolean, and shift unit* processes the data received from the two source buses and gates them to the destination bus. Note again that on a simplified machine, the second source bus may merely be a direct connection from a special register, the accumulator.

All transfers are parallel, and the advantages of having a common word length for transfers between logical units should be obvious.

#### D. Asynchronous Operation

Each logical unit may be referenced asynchronously, and a possible timing will look like the following:

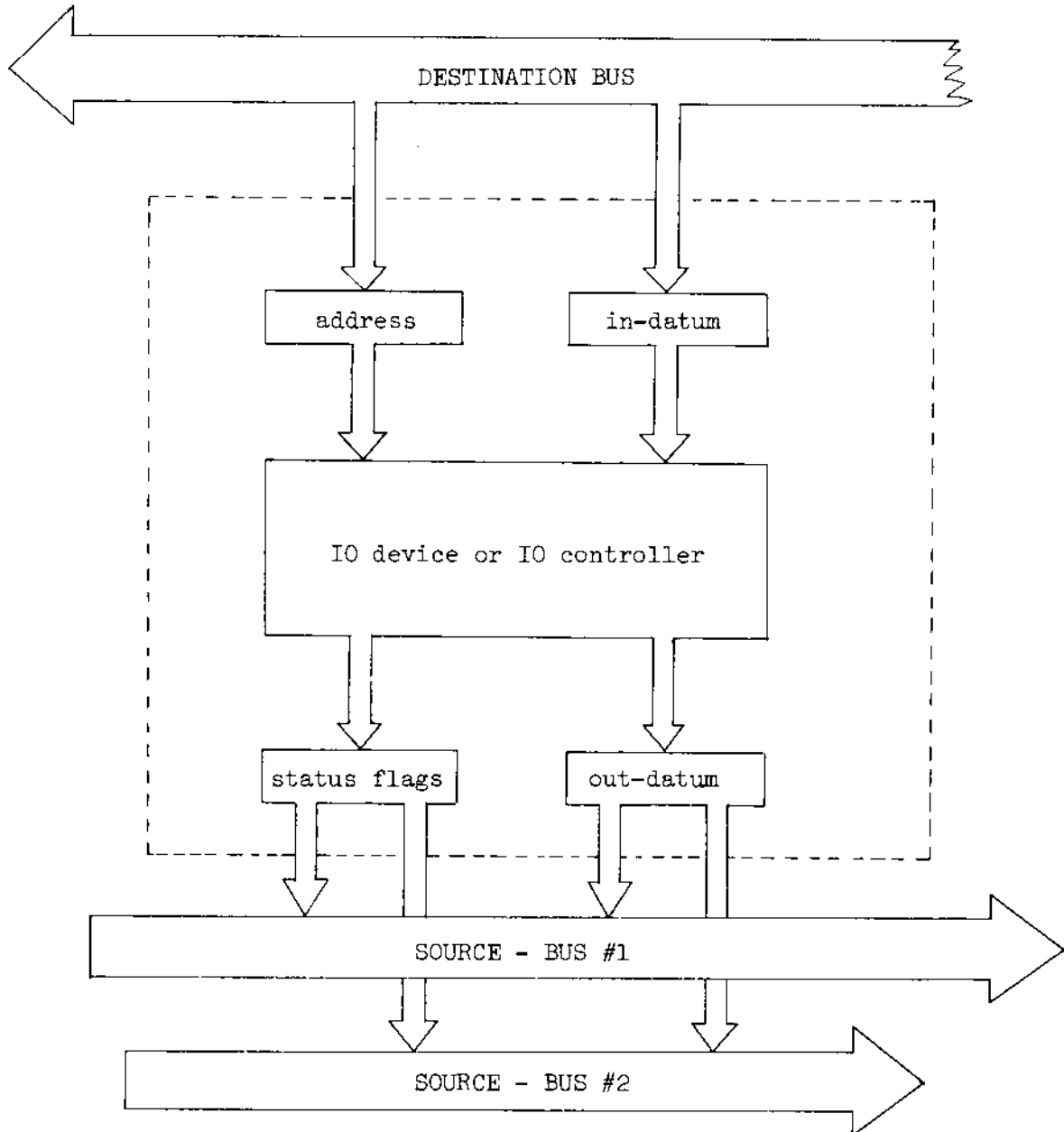
Figure 6.2 - Timing of Micro-operations



Where time elapses downwards and a horizontal line (—) denotes a busy IO or memory unit.

The microprocessor thus makes optimal use of the different timing characteristics of the logical units. It references an IO device, requesting data, and during the time necessary for the data to come back, may initiate a memory access and effect a large number of simple register to register transfers (or branches). Naturally, while the memory unit is busy reading data, another unit higher in the hierarchy may be utilized, and in any case, the fastest unit, the register-unit, may be used a number of times in the interval.

Figure 6.3 - Physical Connections for a Logical IO or Memory Slot



### E. New Developments and Directions

The recent success of microprocessors is mainly due to their efficiency, simplicity and low cost for applications requiring a significant amount of interpretation at the bit level. Microprocessors are naturally used to provide basic processors, that is to interpret a traditional instruction set, and to provide emulated or compatible processors. They are increasingly used to interpret directly a specialized command language and control directly a terminal or IO device. In the case of CRT displays, a significant proportion of these terminals is now equipped with a microprocessor, often dedicated to a unique terminal, in charge of controlling the device and providing sophisticated editing functions and even sometimes controlling a peripheral hard copy device attached to the display.

It can be speculated that, considering the still diminishing cost of such processors, most IO devices will soon be equipped with a dedicated (micro) processor for interpretive control and asynchronous operation.

Another area of great interest is the direct implementation within a dedicated microprocessor of a sophisticated high level language. Little work has yet been done in this direction, and the main published results are those of Melbourne and Pugmire (1965), Weber (1967) and Abrams (1970).

#### F. Language Machines

Although the problem of designing efficient high-level language processors has long been one of the major concerns of system programmers and system designers, it has not been until very recently that it became economically feasible to dedicate a state-of-the-art processor to the processing of a unique high-level language. This accounts for the fact that among the various designs that have been developed over the years for the execution of

high-level languages, very few have attempted direct execution of the language and hardly any report has yet been published on a time-shared or conversational implementation.

As early as 1961, Anderson proposed a machine design for ALGOL 60, followed by FORTRAN designs (Melbourne and Pugmire, 1965), (Bashkow, Sasson and Kronfeld, 1967), an EULER implementation (Weber, 1967) and a theoretical APL machine (Abrams, 1970). These processors are briefly presented:

Melbourne and Pugmire's simulated FORTRAN processor embodies a micro-programmed compiler within control storage. The machine would service one user communicating through a special typewriter, the system's unique input-output device. The console would be equipped with a special keyboard encoding the FORTRAN delimiters and with special keys for debugging and editing. The internal representation is Reverse Polish and all storage references are made in absolute form.

EULER has been developed by Wirth and Weber (1966) as an ALGOL extension and partially implemented as an experimental microprogrammed interpreter on an IBM system 360 model 30. Some features of the language are: dynamic type handling, recursive procedures and dynamic storage allocation, list processing. The interpreter executes reverse Polish strings produced by a microprogrammed translator from the external EULER program. The system is intended to be used by one user in batch or dedicated mode.

Abrams' APL machine presents a method for optimizing the execution of sequences of APL operations on matrices which proves valuable in the case of matrices of large size. This goal is attained by deferring operations

on arrays as long as possible (so as to allow optimization) and by combining all element-by-element operations in an incoming expression into a single operation which is performed once for each element of the result array, thus eliminating many intermediate stores and fetches.



### 3. A System Architecture

A general architecture for real-time APL service is described here. Projections on new architectures based on current hardware developments are presented in the last sections. It must be emphasized that no architecture may be considered an end in itself, but just a step towards a better or new design. It would therefore be unreasonable to claim that the proposed configuration is best for APL time-sharing. Having thus discounted claims to a "best" design, the system is now examined. Details on the original environmental constraints can be found in [Zaks, Steingart, Moore, 1971].

In order to preserve the efficiency of APL execution which characterizes the dedicated microprocessor, it becomes necessary to perform operating system functions on a separate processor. It will be referred to as the Operating System Processor or OSP. The functions of the OSP include traditional Input-Output device management, user command processing, scheduling, editing, translating, diagnostics, file handling. The OSP communicates via a Direct Memory Access (DMA) channel with the APL machine's memory.

The requirement for real-time response imposes on the APL workspace, including the internal Program Strings the constraint of core-residency. This necessitates the use of a large and fast main memory for the APL machine. However, memory cost does not represent any longer a major limitation on the system configuration. In this case, for example, 64K words of 750 nsec core proper are available for less than \$21,000. Also, a special provision has been made within the design of the map for future extension to  $4 \times 64K = 256K$  of memory. The set of address tables necessary to establish the

correspondence between the physical and logical memory locations is contained within a hardware map below. It has been designed integrally with the core memory interface and its access starts at the same time as the decoding of access priority. This results in practically no extra delay during memory access due to the insertion of the map. More details are given in appendix J below.

#### 4. A Language Machine

A hardware processor was selected in 1970 for its cost/performance ratio and a prototype interpreter has been coded into the micro-instruction set of that machine. The basic processor, Digital Scientific Corporation's Meta 4 is a fast microprogrammable processor organized around a well-rationalized three-bus register structure. It may be equipped with up to 32 registers, or 28 registers and a 64-word scratch-pad. The control storage is Read-Only Memory (ROM) and provides 2,000 32 bit micro-instructions. Main memory is here 750 nsec core with up to 64K 18-bit words (16-bit data width). The microprocessor is asynchronous and uses a 3-cycle overlapping technique, cycling in 100 nsec average (Emitter-Coupled Logic). After counting the 80 nsec delay introduced by the two-part core memory interface used here, the actual minimum ratio of micro-instruction executions to memory accesses is 8 to 1.

In order to allocate rapidly and simply the memory in a time-sharing environment, a special page mapping device was designed and built. The map is essentially time-wise invisible during memory accesses, i.e., effects the substitution of a physical address for a virtual one without time overhead. Details of its operation appear in Appendix K.

The vertical microprogramming strategy used in the Meta 4 permits a compact (32 bits) instruction representation and data bus compatibility between registers and ROM. The flexibility of the processor would be still greater if the ROM's data register

were directly accessible on the D-bus, allowing computed micro-instructions to be executed. The micro-instruction set appears in Appendix I. Control instructions are classified in four groups: register to register format (RR), register immediate (RI) where the right word is an immediate operand, branch (BR) and register load (RL) in order to read ROM contents. Branch tests can be effected on any bit, byte or on the whole word. The branch address can be indexed by the Link register. This feature is especially powerful for the construction of jump tables, where the Link contains the index of the table entry. The RR and RI formats allow any of 7 "primitive" arithmetic or boolean operations to be specified. In addition, the RR allows one of 8 shift modifier codes and special controls (jump, carry, decrement counter, plus one, shift in, shift out). Any format may generate memory or IO pulses.

#### 5. Other Processors

Several of the better known mini-computers now available are described in this section. For several of them, it is indicated how efficient APL interpretation could be achieved, using an appropriate configuration.

### Nanodata QM1

This processor is controlled by a 75 nanosecond control store of up to 32K 16-bit words of semi-conductor memory. The main store size is up to 256K words of 750 nanosecond core. There are 32 18-bit internal registers and an extra bank of 32 registers may be added for IO functions and general purpose storage. A special feature of this machine is the fact that a "nanostore" (up to 1K by 360 bit words) establishes the control matrix defining the microinstruction set. The technical information presently available on this processor qualifies it as an excellent candidate to efficient direct language interpretation.

### Microdata - Micro 1600

This processor is byte oriented internally, and accesses a 32K by 16 bit one microsec core memory. Its registers and data paths are eight bit wide. Its control format is 16 bits and the control memory can be expanded to 16K words. It uses internally twelve special purpose registers and a "file" of 30 general purpose 8 bit registers (MSI/LSI). The microcommand execution time is 200 nanoseconds and it has a two address format. The control memory can be implemented in bipolar read-only memory (BROM), programmable read-only memory (PROM) or alterable read-only memory (AROM). AROM permits dynamic microprogramming and convenient debugging. Its limited internal bandwidth and its small microinstruction set appear to qualify this processor for IO and process control better than for language execution.

Lockheed - "Sue"

The SUE computer is a recently introduced highly modular system which permits up to four separate processors to operate within one system. Each processor can be equipped with up to 1K by 36 bit bipolar LSI ROM control memory (60 nanosecond access time). All devices access a common bus, the "Infibus" and a bus transfer is effected in 200 nanoseconds. A separate Infibus controller monitored by the control memory permits data to transit on the bus between memory and peripherals without processor intervention.

The microprocessor obtains its data from 12 general purpose registers on one bus and 4 on a second one. The result is deposited in one of 15 registers. The cycle time is 120 to 160 nanoseconds.

The 36 bit microcode format is subdivided into 13 fields, permitting powerful 3 or 4 address microinstructions.

A four microprocessor configuration of the SUE with a 16 or 32 word high speed scratch pad appears to provide an excellent tool for efficient language execution.

Hewlett Packard - 2100A

This microprogrammed processor may be equipped with up to 1024 by 24 bit words of 196 nanosecond control storage. The control store is configured into four modules of 256 words each (a conventional emulated instruction set requires one module for the microcode). The processor can be equipped with a RAM as Writeable Control Store. Within each module, the 24 bit word is addressed by an 8 bit decoder. The 3 address microinstruction is divided

into six fields: function code, source register A, source register B, destination register C, special field, and skip field.

It may reference nine general purpose registers. The main memory operates at 1 microsec cycle time and can be expanded to 32K sixteen bit words. This processor has a potential for effective language execution by using the writeable control storage for the separable routines. A single processor would not attain the efficiency achievable with 2K or more control storage, due to the loading delay of the RAM. However, a dual processor configuration offers good possibilities for efficient execution.

#### DEC PDP - 11/45 (Digital Equipment Corporation)

The 11/45 CPU has a cycle time of 300 nanoseconds, including instruction fetch. It controls a dual "Unibus" structure and performs arithmetic and logical operations. It operates on sixteen general registers plus six 4 bit floating point accumulators. The Unibus is a common bidirectional and asynchronous communication bus connecting all IO devices through hardware registers. The transfer rate is 2.5 M 16 bit words per second. Bus access is controlled by a priority structure (bus cycle stealing). Although most of the control exercised by the processor is hardwired, its structure and instruction set relate it to microprogrammable processors.

Three types of memory are available: 850 nanosecond core, 300 nanosecond bipolar, 450 nanosecond MOS. Memory can be expanded from 4K to 126K in 4K increments. Maximum solid state is 32K.

Communication between the CPU and the solid state memory is through a high speed data path internal to the processor. It does not use the Unibus. The CPU can control two independent solid state memory controllers (up to 16K each). While the CPU accesses one bank, the other one may be accessed by an IO device through the Unibus.

Excellent floating point facilities are provided: single and double precision (8 bit exponent, 55 bits + sign, fraction), 17 digit accuracy. The average execution time for a double precision floating point multiply is 8.5 microseconds.



## 6. Performance

The performance of the interpreter is now evaluated on some sample runs. The values obtained are typical, but could be reduced by further refinement of the routines. The first case considered is the addition of two vectors: 1 2 3 4 5 + 2 3 4 5 6. In order to provide a conservative estimate, the memory is assumed to cycle in 1 $\mu$ . The essential delays are tabulated in figure 6.4 below. The complete timing is given in Appendix N . The total time necessary for line execution (up to print statement) is 113.4 $\mu$ sec.

As can be observed on figure 6.4, scanning a token and identifying its syntactic type takes 1.4 $\mu$ sec. In the case of phantoms, this operation requires 4.5 $\mu$ sec because of the additional table look up which must be performed. For a variable, the overhead for run-time resolution of phantoms can therefore be seen to be 3.1 $\mu$ sec.

Figure 6.4 - Timings for Addition of Two 5-element Arrays

Action	time in $\mu$ 's
Start line	3.9
Parse Next Token	1.0
Decypher: array	.4
Change parser state-checks	2.2
Parse Next Token	1.0
Decypher: operator	.4
Change parser state-checks	.9
Parse Next Token	1.0
Decypher: array	.4
syntax check	1.6
Binary Operator. Operator front end. identify VECVEC	2.0
Array front end: conformability, getblock, initialize	25.4
add and store 5 scalars	60.5
Operator exit	10.2
end line	2.5

Figure 6.5 - Timings in Array-Array Operations

Overhead: start  $\simeq 19 \mu\text{s}$

exit  $\simeq 30 \mu\text{s}$

Timings are for one cycle involving the operator.

Operator	Time for One Operation ( $\mu\text{s}$ )
$\times$	27.6
$\div$	33.1
$\vee$	7.5
$\wedge$	10.4
$\sim$	11.8
$\neg$	11.1
$=$	13.1
$\#$	14.1
$\leq$	13.3
$\geq$	16.7
monadic: +	6.5
$\perp$	20.0
$\lceil$	25.0
$\div$	23.0

In order to evaluate the overhead associated with the Function Reference Zone mechanism, the timing of a recursive function is examined.

It is the recursive version of the factorial function:

```

VZ ← FAC N
[1] → 4 × 1 N
[2] Z ← N × FAC N - 1
[3] → 0
[4] Z ← 1
▽

```

The function is called successively with

```

FAC 9
FAC 8
FAC 7
FAC 6
FAC 5
FAC 4
FAC 3
FAC 2
FAC 1
FAC 0

```

The printout of the actual run appears in appendix O. The time necessary to execute FAC 9 can be analyzed as follows:

- initial parsing up to the function call ; time  $t_i$
- successive recursions ( $N + 1$  function calls).

The time spent in level  $m$  is  $r_m$ . Time  $r_0$  for FAC 0 is longer than other  $r_i$ 's because of the different branching within the function body.

- final exit time  $t_f$ .

Let  $r_p$  be the time from entry to return, at level  $p$ . As can be seen on page 0-9 of appendix 0, the average value for one recursion level is  $r = 165 \mu\text{sec}$ .

A good approximation for the time necessary to execute FAC  $N$  is therefore:

$$t = t_i + r_0 + Nr + t_f$$

or  $t = 9.4 + 192 + 165N + 15.8 \text{ in } \mu\text{s}.$

or  $\underline{t_{\mu\text{s}} = 217 + 165N}$

The equation is tabulated below against the actual measurements:

N	Computed Time	Actual
1	382	381
2	547	547
3	712	712
9	1702	1706

Having established a global timing equation for comparison with other implementations, the internal delays are now scrutinized more closely. As can be seen on page 0-3 of appendix 0, the parser identifies a phantom in  $1.5 \mu\text{s}$  and recognizes a function call in an additional  $2.9 \mu\text{s}$ .

The syntax is verified in 5.0 us (timings 98 to 148 on 0-3). At time 148 on 0.3 (time is in tenths of microseconds), the NUFRAME routine is reached. A new frame is created on the Stack (new LFRZ). It has two local variables and no labels. The LFRZ is created in 5.7 us. It can be seen that the time required for each additional local variable is .6 us in order to enter a UV descriptor in the LFRZ. This is less than one core cycle time even though it involves a write operation. This is a direct benefit of the parallelism inherent to the microprocessor. Core access and microoperations proceed in parallel. Finally, the new history segment (NUHISTORY) is created on top of the LFRZ in 10.6 us. Again this involves the retrieval and storage of seven words and is accomplished in only slightly longer than it takes for the memory to cycle.

Finally, there are no labels in this case (ENTERLABELS) and pointers are initialized for the first line of the function. The whole process, from FCNCALL to STARTLINE has taken 22.8 us. During this time, 4 memory-read and 7 memory write-operations have been performed. The memory has therefore been cycling for 11 us.

Similarly, the return operations are apparent on page 0-6. The LFRZ is scanned for possible array values (to be released). The cost is 1.6 us per entry. Finally, history pointers are restored and the result deposited on top of the stack in approximately 9 us. Parsing then resumes.

In summary, the cost of the bookkeeping operations associated with the function mechanism is analogous to the cost of executing an APL operator. It increases slowly as the number of variables and arrays increases. During these management operations, advantage is taken of the processing capability of the microprocessor during memory accesses and the total overhead is usually less than twice the active time of the core.

Figure 6.6 - Size of the Routines (# of instructions)

G L O B A L		O P E R A T O R S	
Utility Routines	207	front end	276
Parser (total)	583	return	39
Operators Front end & Return	315	fix	16
Microprogrammed Operators	523	normalize	23
Indexing	182	dyadic: -	16
		+	59
		x	31
		÷	36
		Γ	13
		L	9
			33
		^	4
		^	4
		✓	4
		✓	4
		<	6
		≤	6
		≥	2
		>	2
		=	17
		#	4
		p	119
		monadic: -	6
		+	0
		x	4
		÷	3
		Γ	21
		L	2
			3
		~	4
		2	72
TOTAL	1,810	TOTAL	838

P A R S E R	
general	88
function call	119
operator	78
branch	39
semicolon	9
function return	46
leftbrace	2
leftparen	23
leftbracket	9
line delimiter	18
assignment	97
null	4
rightbrace	36
printAR	12
parameter	3
TOTAL	583



## 7. Conclusions

It is in the nature of an interpreter to require a high ratio of elementary logical and arithmetic operations to individual data fetches. This ratio is limited to 1 by the structure of a conventional processor, but does not have any intrinsic limitation on a microprogrammed processor. It can even be adjusted on a microprocessor by modifying memory type (speed) or processor speed, where possible. A high level language interpreter can take full advantage of this hardware characteristic. However, it was noted that a technological limitation stems from the control storage deficiency of current fast microprocessors, typically less than 2,000 instructions. In the course of engineering such an interpreter, a conceptual analysis of the properties of APL allowed a systematic approach to a terse parser design for a class of languages called Nested Primitive Languages and to a simple design of Virtual Memory structures for APL.

As an additional benefit, the discipline applied to the construction of the interpreter has resulted in a high internal consistency. For example, in the case of subexpressions containing (middle-of-line) assignments, the order of evaluation is guaranteed to be right-to-left, and not undeterminate as in APL/360. Also labels are not subject to special restrictions and are simply initialized local variables. They may be assigned new values during execution and are treated like any local variable. This differs from APL/360 where restrictions on the use of labels hold (no assignments to a label, but any variable may be used as a label in a branch statement).

These results constitute only the beginning of an investigation into the properties of APL or APL-like languages. They may prove valuable for further research in the structure of programming languages and their implementation.

Consequences of these results in the field of computer systems architecture can also be considered. As is apparent from the measurements presented in section 6, a microprogrammed APL implementation rivals in speed much larger computer systems. In fact, the efficiency is sufficiently high to consider for purposes of cost reduction to have a complete APL system reside on a unique processor. This naturally implies the availability of a control storage which can be expanded physically (large storage) or virtually (read-write storage). This approach limits the response-time and the number of users among which the processor may be time-shared. The extreme application of this philosophy opens the door realistically to a new possibility: a desk-top APL calculator. A microprogrammed APL system for a single user can now be implemented directly within the control storage of a contemporary microprocessor-controlled CRT ( $\mu$ CRT) display terminal. The hardware cost of this APL-CRT one user system is then only the cost of a standard  $\mu$ CRT plus the additional memory for the interpreter.

Exploring now the opposite direction, it can be predicted that the ever-decreasing costs of processor manufacturing will result in CPU costs much inferior to the one of an input-output device. It is therefore becoming increasingly feasible to dedicate a computing element per execution function. Carrying this approach further, the concept of distributed power can be applied to all

distinguishable routines within the interpreter. In order to increase the execution efficiency in a time-sharing environment, each separable function can therefore reside on a dedicated processor. This approach is not limited by the complexity of time-sharing multiple processors efficiently for one or more users, since the system operates in the context of a single language.

Finally, a further gain in speed can be achieved by implementing directly in hardware those algorithms which become completely defined and will not be modified, such as floating point arithmetic routines.

## **A P P E N D I C E S**

- A - The Syntax of a Primitive Programming Language in BNF
- B - Simplified Syntax for Meta-APL in BNF
- C - A Phantom Processor ( $\Phi$ -Processor)
- D - The Translator
- E - Listing of the Translator in Snobol
- F - Representation of APL Operators
- G - Listings of the Software APL Operators
- H - Table of Floating Point Integers
- I - Meta 4 Control Instructions
- J - The Hardware Map
- K - Map Control
- L - Input-Output Registers
- M - Listing of the Microprogrammed Interpreter
- N - Complete Timing for Addition
- O - Timing of a Recursive Function

## THE SYNTAX OF A PRIMITIVE PROGRAMMING LANGUAGE IN BNF

## Appendix A

## The Syntax of a Primitive Programming Language in BNF

1.1 <STATEMENT> ::= <EXPRESSION>

1.2 <EXPRESSION> ::= <OPERAND-TYPE>

| {<EXPRESSION>} <FUNCTION-TYPE> <EXPRESSION>

Operand-types and function-types can be defined in turn in many ways.

For example:

<OPERAND-TYPE> ::= <VARIABLE> | <NUMERICAL TYPE>

<FUNCTION-TYPE> ::= <OPERATOR> | <FUNCTION>

with:

<VARIABLE> ::= <NAME>

<NUMERICAL TYPE> ::= <NUMBER> | <VECTOR>

<FUNCTION> ::= <NAME>

<OPERATOR> ::= <MONADIC OPERATOR> | <DYADIC OPERATOR>

and:

```

<NUMBER> ::= <DECIMAL FORM> | <EXPONENTIAL FORM>

<DECIMAL FORM> ::= {<INTEGER>}{•<INTEGER>}

<INTEGER> ::= <DIGIT> | <INTEGER><DIGIT>

<DIGIT> ::= Ø | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<EXPONENTIAL FORM> ::= <DECIMAL FORM>E<INTEGER>

<VECTOR> ::= <SCALAR VECTOR> | <CHARACTER VECTOR>

<SCALAR VECTOR> ::= <NUMBER> | <SCALAR VECTOR><SPACES><NUMBER>

<SPACES> ::= <SPACE> | <SPACES><SPACE>

<SPACE> ::= <ONE BLANK>

<CHARACTER VECTOR> ::= '<CHARACTER STRING>'

<CHARACTER STRING> ::= <CHARACTER> | <CHARACTER STRING><CHARACTER>

<CHARACTER> ::= <LETTER> | <DIGIT> | <SYMBOL>

<NAME> ::= <LETTER> | <LETTER> <ALPHANUMERIC STRING>

<ALPHANUMERIC STRING> ::= <ALPHANUMERIC> | <ALPHANUMERIC STRING>
                                <ALPHANUMERIC>

<ALPHANUMERIC> ::= <LETTER> | <DIGIT>

<LETTER> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N
            O | P | Q | R | S | T | U | V | W | X | Y | Z

<SYMBOL> ::= ] | [ | + | → | + | × | / | \ | , | . | " | ' | < | > | ≠ | ≤ | ≥ | = | ) | (
            √ | ∧ | € | : | ⊥ | T | ⊥ | + | † | ‡ | ~ | Ø | ? | L | Γ | - | * | ρ | U | ∩
            | α | C | D | ÷ | ω | □ | ○ | ∇ | Δ | ' | { | } | |

```

SIMPLIFIED SYNTAX FOR META-APL IN BNF



## Appendix B

## Simplified Syntax for Meta-APL in BNF

- Notes: (1) {...} denotes  $\emptyset$  or 1 times .... {} are symbols of Meta-APL used to flag function parameters.
- (2) Lower-case letters are used for comments to avoid lengthy repetitions.
- (3) cr denotes a carriage-return.
- (4) PROGRAM in BNF is equivalent to PROCESS in the text.
- (5) wner means "with no explicit result". Unless wner is used, a <0,1 or 2-ARG FUNCTION> has an explicit result.

## Group 1

```

<FUNCTION BLOCK>      ::= <FUNCTION DEFINITION>cr<STATEMENTS> v
<PROGRAM BLOCK>       ::= {<PROGRAM DEFINITION>}cr<PROGRAM>
<PROGRAM DEFINITION>  ::= <PROGRAM NAME>{{<PARAMETER LIST>}}
<PROGRAM NAME>        ::= <NAME>
<PROGRAM>             ::= <STATEMENT> | <PROGRAM><STATEMENT>
                        | <PROGRAM><FUNCTION BLOCK>
<STATEMENTS>          ::= <STATEMENTS> <STATEMENT>
<STATEMENT>           ::= {<LABEL>}<STATEMENT LINE>cr
<STATEMENT LINE>      ::= <BRANCH> | <SPECIFICATION> | <SYSTEM COMMAND>
                        | <IMMEDIATE> | <ROUTINE>
<SYSTEM COMMAND>      ::= <see system commands>
<BRANCH>              ::= →<EXPRESSION>
<IMMEDIATE>           ::= <EXPRESSION> | <EXPRESSION>;<IMMEDIATE>
<SPECIFICATION>       ::= <VARIABLE>←<EXPRESSION> | <OUTPUT SYMBOL>
                        ←<EXPRESSION>
<ROUTINE>             ::= <∅ ARG FUNCTION wner> | <1-ARG FUNCTION wner>
                        | <2-ARG FUNCTION wner>

```

## Group 2

```

<NUMBER>              ::= <DECIMAL FORM> | <EXPONENTIAL FORM>
<DECIMAL FORM>        ::= {<INTEGER>}{•<INTEGER>}
<INTEGER>             ::= <DIGIT> | <INTEGER><DIGIT>
<DIGIT>              ::= ∅ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<EXPONENTIAL FORM>    ::= <DECIMAL FORM>E<INTEGER>
<VECTOR>              ::= <SCALAR VECTOR> | <CHARACTER VECTOR>
                        | <EMPTY VECTOR>
<SCALAR VECTOR>       ::= <NUMBER> | <SCALAR VECTOR><SPACES><NUMBER>
<SPACES>              ::= <SPACE> | <SPACES><SPACE>

```

<SPACE>	::= <one blank>
<EMPTY VECTOR>	::= " $1\emptyset$ " <SCALAR>
<CHARACTER VECTOR>	::= '<CHARACTER STRING>'
<CHARACTER STRING>	::= <CHARACTER>   <CHARACTER STRING> <CHARACTER>
<CHARACTER>	::= <LETTER>   <DIGIT>   <SYMBOL>
<NAME>	::= <LETTER>   <LETTER> <ALPHANUMERIC STRING>
<ALPHANUMERIC STRING>	::= <ALPHANUMERIC>   <ALPHANUMERIC STRING> <ALPHANUMERIC>
<ALPHANUMERIC>	::= <LETTER>   <DIGIT>
<NUMERICAL TYPE>	::= <NUMBER>   <VECTOR>
<LABEL>	::= <NAME> :
<IOSYMBOL>	::= <INPUT SYMBOL>   <OUTPUT SYMBOL>
<OUTPUT SYMBOL>	::= $\square$   <DEVICE ID>
<INPUT SYMBOL>	::= $\square$   $\square$
<DEVICE ID>	::= <undefined as yet>

## Group 3

<SCALAR OPERATOR>	::= <MONADIC SCALOP>   <DYADIC SCALOP>
<MONADIC OPERATOR>	::= <MONADIC SCALOP>   <MONADIC MIXEDOP>   <MONADIC EXTENDED SCALOP>
<DYADIC OPERATOR>	::= <DYADIC SCALOP>   <DYADIC MIXEDOP>   <DYADIC EXTENDED SCALOP>
<MONADIC SCALOP>	::= +   -   $\times$   $\div$   $\Gamma$   $\perp$   *   $\otimes$     !   ?   $\bigcirc$   $\sim$
<DYADIC SCALOP>	::= <MONADIC SCALOP>   $\wedge$   $\vee$   $\wedge$   $\vee$   $\leq$   $\geq$   $>$   $\neq$
<EXTENDED SCALAR OPERATOR>	::= <MONADIC EXTENDED SCALOP>   <DYADIC EXTENDED SCALOP>
<MONADIC EXTENDED SCALOP>	::= <SCALAR OPERATOR> /   <INDEXED OPERATOR> / [ <EXPRESSION> ]
<DYADIC EXTENDED SCALOP>	::= <DYADIC SCALOP> • <DYADIC SCALOP>   0 • <DYADIC SCALOP>

$\langle \text{MIXED OPERATOR} \rangle ::= \langle \text{DYADIC MIXEDOP} \rangle | \langle \text{MONADIC MIXEDOP} \rangle$   
 $\langle \text{MONADIC MIXEDOP} \rangle ::= \rho | , | \sim | \phi | \otimes | \Delta | \Psi | \nabla | \Delta$   
 $\langle \text{DYADIC MIXEDOP} \rangle ::= \rho | , | \iota | \phi | \otimes | / | \backslash | \dagger | \ddagger | \in | \perp | \top |$   
 $\langle \text{INDEXED OPERATOR} \rangle ::= / | \phi | \backslash$

## Group 4

$\langle \text{EXPRESSION} \rangle ::= \langle \text{NUMERICAL TYPE} \rangle | \langle \text{VARIABLE} \rangle | \langle \text{INPUT SYMBOL} \rangle$   
 $\quad | \langle \emptyset\text{-ARG FUNCTION} \rangle | \langle \text{MONADIC EXPRESSION} \rangle$   
 $\quad | \langle \text{DYADIC EXPRESSION} \rangle | \langle \text{SPECIFICATION} \rangle$   
 $\quad | ( \langle \text{EXPRESSION} \rangle ) | \langle \text{INDEXED EXPRESSION} \rangle$   
 $\langle \text{MONADIC EXPRESSION} \rangle ::= \langle \text{MONADIC OPERATOR} \rangle \langle \text{EXPRESSION} \rangle$   
 $\quad | \langle \text{1-ARG FUNCTION} \rangle$   
 $\langle \text{DYADIC EXPRESSION} \rangle ::= \langle \text{EXPRESSION} \rangle \langle \text{DYADIC OPERATOR} \rangle \langle \text{EXPRESSION} \rangle$   
 $\quad | \langle \text{ALPHANUMERIC STRING} \rangle \langle \text{RELOP} \rangle$   
 $\quad \langle \text{ALPHANUMERIC STRING} \rangle | \langle \text{2-ARG FUNCTION} \rangle$   
 $\langle \text{RELOP} \rangle ::= < | \leq | = | \geq | > | \neq$   
 $\langle \text{FUNCTION NAME} \rangle ::= \langle \text{NAME} \rangle$   
 $\langle \emptyset\text{-ARG FUNCTION} \rangle ::= \langle \text{FUNCTION NAME} \rangle \{ \{ \langle \text{PARAMETER LIST} \rangle \} \}$   
 $\langle \text{1-ARG FUNCTION} \rangle ::= \langle \text{FUNCTION NAME} \rangle \{ \{ \langle \text{PARAMETER LIST} \rangle \} \}$   
 $\quad \langle \text{EXPRESSION} \rangle$   
 $\langle \text{2-ARG FUNCTION} \rangle ::= \langle \text{EXPRESSION} \rangle \langle \text{FUNCTION NAME} \rangle$   
 $\quad \{ \{ \langle \text{PARAMETER LIST} \rangle \} \} \langle \text{EXPRESSION} \rangle$   
 $\langle \text{FUNCTION DEFINITION} \rangle ::= \nabla \{ \langle \text{VARIABLE NAME} \rangle \} \{ \leftarrow \{ \langle \text{VARIABLE NAME} \rangle \}$   
 $\quad \langle \text{FUNCTION NAME} \rangle \{ \{ \langle \text{PARAMETER LIST} \rangle \} \}$   
 $\quad \langle \text{VARIABLE NAME} \rangle \{ \langle \text{LOCAL VARIABLES} \rangle \}$   
 $\quad | \nabla \{ \langle \text{VARIABLE NAME} \rangle \} \leftarrow \langle \text{FUNCTION NAME} \rangle$   
 $\quad \{ \{ \langle \text{PARAMETER LIST} \rangle \} \} \{ \langle \text{VARIABLE NAME} \rangle \}$   
 $\quad \{ \langle \text{LOCAL VARIABLES} \rangle \}$   
 $\langle \text{VARIABLE} \rangle ::= \langle \text{NAME} \rangle$   
 $\langle \text{INDEXED EXPRESSION} \rangle ::= \langle \text{EXPRESSION} \rangle [ \langle \text{INDEXING LIST} \rangle ]$   
 $\langle \text{INDEXING LIST} \rangle ::= \{ \langle \text{EXPRESSION} \rangle \} \{ ; \langle \text{INDEXING LIST} \rangle \}$   
 $\langle \text{PARAMETER LIST} \rangle ::= \langle \text{PARAMETER NAME} \rangle | \langle \text{PARAMETER LIST} \rangle$   
 $\quad ; \langle \text{PARAMETER NAME} \rangle$

```

<LOCAL VARIABLES>      ::= ;<VARIABLE NAME>
                           |<LOCAL VARIABLE>;<VARIABLE NAME >

<PARAMETER NAME>       ::= <VARIABLE NAME>

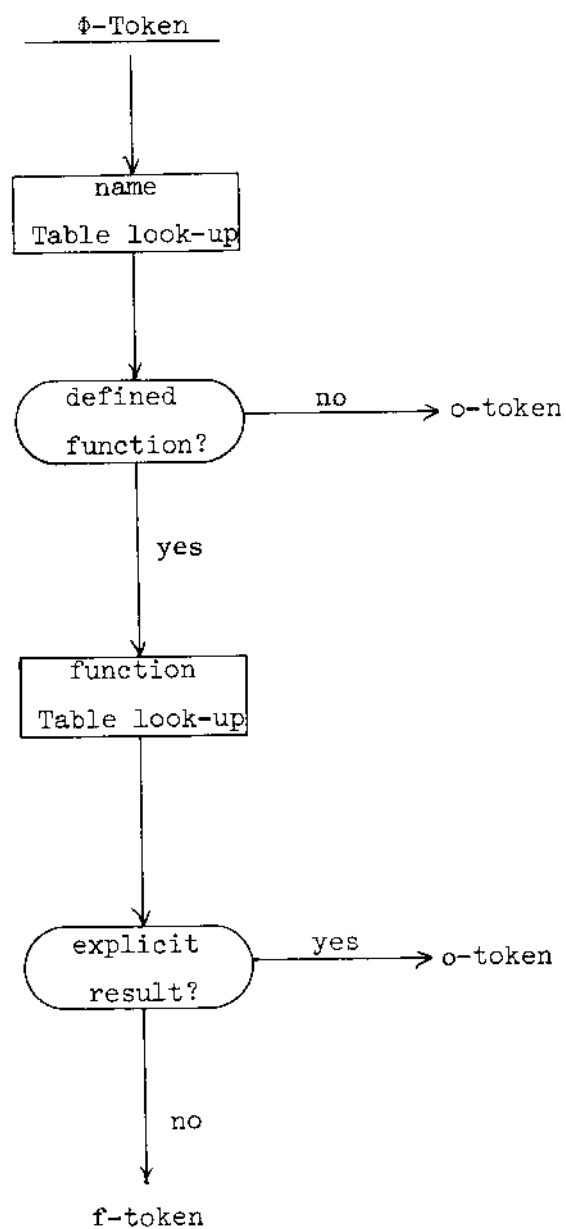
<LETTER>                ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N
                           O|P|Q|R|S|T|U|V|W|X|Y|Z

<SYMBOL>               ::= ]|[|+|→|+|×|/|\|,|.|'|-|<|>|≠
                           |≤|≥|=|)|(|(|V|∧|e|:|⊥|⊤|_|+|↑|↓
                           |~|o|?|L|Γ|-|*|o|U|∩|α|C|D|÷|ω
                           □|o|∇|Δ|'|{|}|

```

A PHANTOM PROCESSOR

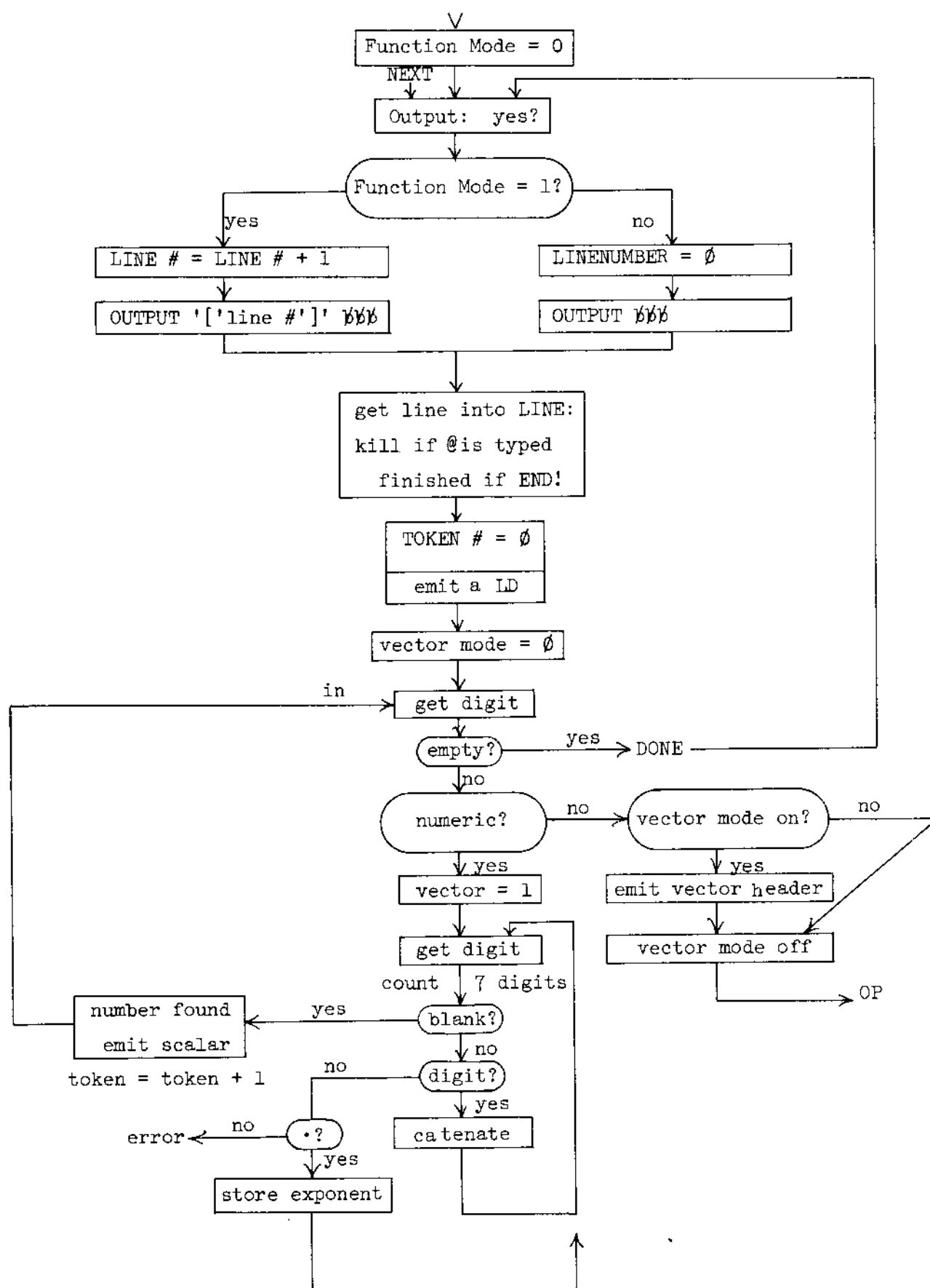
## Appendix C

A Phantom Processor ( $\Phi$ -Processor)

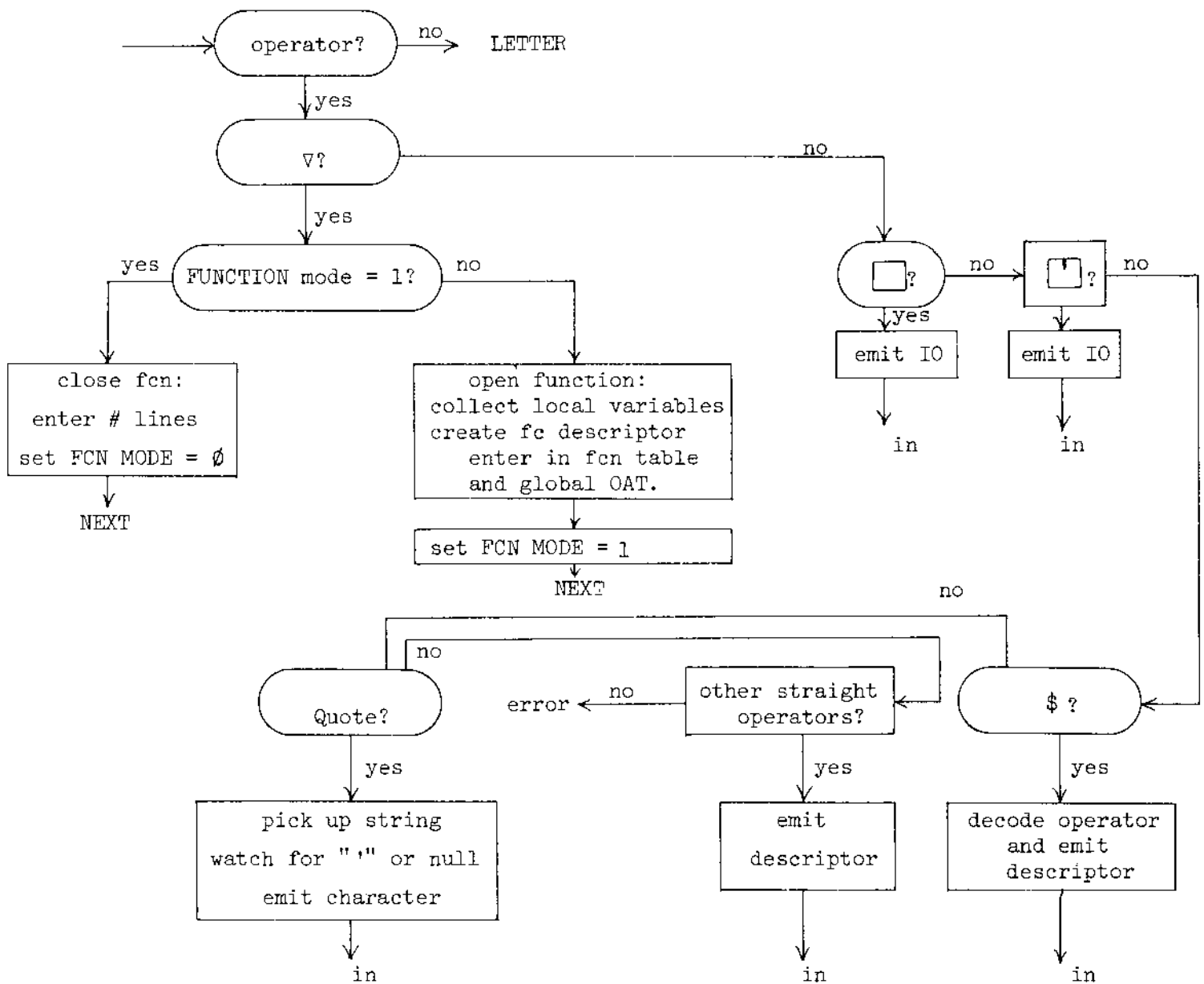
## THE TRANSLATOR

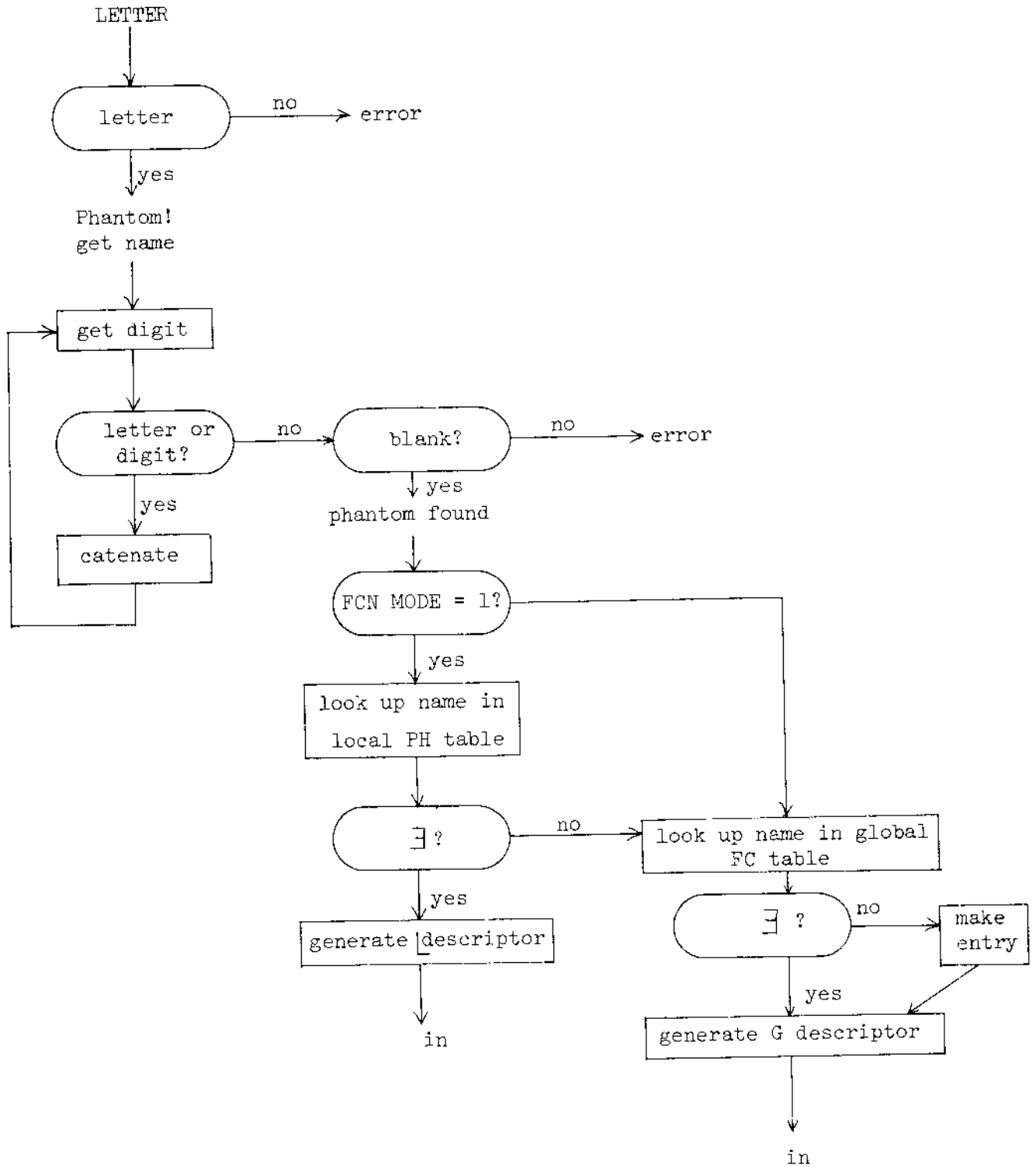


## Appendix D

THE TRANSLATOR

## Appendix D

The Translator (continued)

The Translator (continued)

## LISTING OF THE TRANSLATOR IN SNOBOL

## Appendix E - Listing of the Translator in Snobol

```

*CODES
*****
*
START  GPHCODE      = 114000
        LPHCODE      = 110000
        FCCODE       = 160000
        VCODE        = 100000
        LDOPCODE     = 122004
*
*****
*PATTERNS**
*****
*
      NAME = ANY(@ALPHABET) (SPAN(@ALPHABET @DIGITS) ! ' ' )
      OP1 = ANY("( )+-=#$,%'")
      OP2 = ANY("' /\ * ! ? . > < -")
      OP3 = '$GE' ! '$LE' ! '$TIM' ! '$DIV' ! '$AND' ! '$OR' ! '$NOT' &+
            ! '$NAND' ! '$NOR' ! '$IOP' ! '$ROT' ! '$BAR' ! '$PI' &+
            ! '$LOG'
      OP4 = '$IOTA' ! '$RRD' ! '$ELMT' ! '$TAKE' ! '$DROP' &+
            ! '$ROI' ! '$TRANS' ! '$UP' ! '$DOWN' ! '$DEC' &+
            ! '$ENC' ! '$EXP0' ! '$0' ! '$GOTO' ! '$QUAD' &+
            ! '$NEG' ! '$RERACE' ! '$LRACE'
      OPERATOR = OP1 ! OP2 ! OP3 ! OP4
      INTEGER = SPAN@DIGITS
      FIGNUMBER = INTEGER ('.' ! ' ') (INTEGER ! ' ')
*THESE ARE THE FUNCTIONS:
*x IS A DECIMAL NUMBER
*
      DEFINE('OCTAL(x)')
      DEFINE('BIN(x)')
      DEFINE('OCTBIN(x)')
*
*x IS AN INTEGER TO BE FLOATED.NO LEADING ZEROES
      DEFINE('FLOAT(x)')
      DEFINE('COMP(x)')
*
*
*

```

```

*****
*TRANSLATOR*
*****
*
      GPHANTOMNBR = -1
      LPHANTOMNBR = -1
      GLOBALBASE  = 4009
      LABELBASE   = 600
      NBROFLINES  = 0
      LIST =
      TOTALTOKENS = 0
      OUTPUT =
      LTRASE = 532
      OUTPUT =
      OUTPUT =
GETBASE BASE = 650
      OUTPUT =
      INTEGER(BASE)          : F(GETBASE)
*BASE FOR :      CORE:WADDR, BASE: STRINGS...
      OUTPUTC = 'FCNBR START AT ... = '
      FCNBR = INPUT
      FUNCTIONMODE = 0
*
*
*
      OUTPUT = '      ' DATE() '      ' TIME()
      OUTPUT =
      OUTPUT = "TYPE '0' TO DELETE A LINE  "
      OUTPUT = "TYPE 'END' WHEN FINISHED"
      OUTPUT =
      OUTPUT =
*OPEN OUTPUT FILE
      P = OPENOUT('TESTK:'):
      ASSIGN('OUT',P)
      OUTPUT = 'META-APL TRANSLATOR READY...'
      OUTPUT =
      OUTPUT =
*PROGRAM MODE:
PROGRAM LINENBR = 0
AGAIN EFC(FUNCTIONMODE,1)          : S(FCNMODE)
      OUTPUTC = '      '
CON      TOKENNBR = 0
      BASE = BASE + TOTALTOKENS
      STRING =
      TOTALTOKENS = 0
*EACH LINE ENDS WITH AN LD. GENERATE IT.
      STRING = LDJFCODE 'B'
      TOTALTOKENS = TOTALTOKENS + 1
      VECTORMODE = 0          : (IN)
FCNMODE LINENBR = LINENBR + 1
      OUTPUTC = '      ' [' LINENBR ']' '      '          : (CON)
*

```

```

*BASIC INPUT LOOP:
IN.   LINE = TRIM(INPUT)
      LINE '0'                                     :F(ACCEPTED)
      LINENBR = G(LINENBR,0) LINENBR = 1
      OUTPUT = 'LINE DELETED'
      OUTPUT =                                     : (AGAIN)
ACCEPTED LINE POS(0) 'END' RPOS(0)                 :S(ENDF)
      OUT = '*' LINE
      ELEMENTNBR = 0
      @FULLSCAN = -1
      LINE '...' = '...'
      LINE '....' = '....'
      @FULLSCAN = 0
      LINE POS(0) ARE . LABEL ':' =                :S(LABELFOUND)
TRIMIT LINE POS(0) ' ' =                           :S(TRIMIT)
      RETURNPT = 'TRIMIT'
      IDENT(LINE,) EQ(VECTORMODE,1) G(ELEMENTNBR,1) :S(VEC)
      IDENT(LINE,)                               :S(EMIT)
*TEST FOR INTEGER:
      LINE POS(0) FIGNUMBER . NUMBER =             :F(NONUMBER)
      SIGN = 0
*POSITIVE NUMBER FOUND:
NUMBEROK VECTORMODE = 1
*COUNT # ELEMENTS FOR A VECTOR
      ELEMENTNBR = ELEMENTNBR + 1
COUNTIT TOKENNBR = TOKENNBR + 1
      TOTALTOKENS = TOTALTOKENS + 1
      RETURNPT = 'TRIMIT'
      CODE = FLOAT(NUMBER)                          : (GENERATE)
NONUMBER LINE POS(0) 'SNEG' =                      :S(NEG)
      NE(VECTORMODE,1)                              :S(NAME)
      EQ(ELEMENTNBR,1)                              :S(NOVECTOR)
*IT IS A VECTOR. CREATE THE HEADER:
**# ELEMENTS:
VEC    VECTORMODE = 0
      RETURNPT = 'LAB1'
      CODE = OCTAL(ELEMENTNBR)                      : (GENERATE)
* DEFERMENT FIELD:
LAB1   CODE = '000000'
      RETURNPT = 'LAB2'                            : (GENERATE)
*TOTAL # ELEMENTS:
LAB2   CODE = OCTAL(ELEMENTNBR)
      RETURNPT = 'LAB3'                            : (GENERATE)
*SIZE
LAB3   CODE = OCTAL(ELEMENTNBR * 2 + 5)
      RETURNPT = 'LAB4'                            : (GENERATE)
LAB4   CODE = '100001'
      ELEMENTNBR = 0
      RETURNPT = 'TRIMIT'                          : (GENERATE)
*
NOVECTOR ELEMENTNBR = 0
      VECTORMODE = 0                                : (NAME)
*
*
```

```

NAME      LINE POS(0) 'SDEL' =                               :S(DEL)
          VECTORMODE = 0
          PHANTOM =
          LINE POS(0) NAME . PHANTOM =                         :F(OP)
          N = -1
*N IS PHANTOM #
          EQ(FUNCTIONMODE,1)                                   :F(GLOBALLOOKUP)
LOCALLOOKUP      N = N + 1
                  IDENT(S('LPHANTOM' N), PHANTOM) :S(LOCALPHFOUND)
                  LT(N,LPHANTOMNBR)                :S(LOCALLOOKUP)F(GLOBAL)
LOCALPHFOUND      CODE = (LPHCODE + OCTAL(N)) :((GENERATE))
GLOBAL      N = -1
GLOBALLOOKUP      N = N + 1
                  BACK = 'GLOBALPHFOUND'
                  IDENT(S('GPHANTOM' N),PHANTOM) :S(GLOBALPHFOUND)
                  LT(N,GPHANTOMNBR)                :S(GLOBALLOOKUP)
CREATEGPH      GPHANTOMNBR = GPHANTOMNBR + 1
*CREATE NEW ENTRY IN GLOBAL TABLE:
                  N = GPHANTOMNBR
                  S('GPHANTOM' GPHANTOMNBR) = PHANTOM :((SBACK))
GLOBALPHFOUND      CODE = (GPHCODE + OCTAL(N))
                  RETURNPT = 'TRIMIT' :((GENERATE))
*
LABELFOUND LABEL POS(0) ' ' :S(LABELFOUND)
          LABEL = TRIM(LABEL)
          LABEL POS(0) NAME . PHANTOM RPOS(0) :F(ERROR1)
          LABELNBR = LABELNBR + 1
          S('LABEL' LABELNBR) = PHANTOM ' = ' LINENBR ' (LINE #) '
          LABELSTRING = OCTAL(LPHANTOMNBR + 1) 'B,' FLOAT(LINENBR) 'F,' &
          LABELSTRING
          BACK = 'TRIMIT' :((CREATELPH))
*
```



```

OP      LINE POS(0) OPERATOR = OPR      =      :F(ERROR1)
        VEQFORMODE = 0
        RETURNPI = 'TRIMIT'
*OPERATOR FOUND:
        OPR POS(0) '$' REM = OPLABEL      :S($OPLABEL)
*NO $ OPERATOR
        OPR '+'      :S(PLUS)
        OPR '-'      :S(MINUS)
        OPR '*'      :S(STAR)
        OPR '<'      :S(LESS)
        OPR '='      :S(EQUAL)
        OPR '#'      :S(NEQUAL)
        OPR '>'      :S(GREATER)
        OPR '!'      :S(XPNT)
        OPR '?'      :S(OMARK)
        OPR '/'      :S(SLASH)
        OPR ','      :S(COMMA)
        OPR '['      :S(LBRKT)
        OPR ']'      :S(RBRKT)
        OPR '^'      :S(LARROW)
        OPR '"'      :S(QUOTE)
        OPR '('      :S(LPAREN)
        OPR ')'      :S(RPAREN)
        OPR ';'      :S(SEMICLN)
        OPR '.'      :S(INNERPRT)
ERROR OUTPUT = OPR 'IS ILLEGAL OPERATOR. TRY AGAIN ...':(AGAIN)
ERROR3 OUTPUT = 'SYSTEM ERROR'      : (END)
*
*
*
```

\*THE OPERATOR CODES :  
 \*=====

\*  
 \*

PLUS CODE = 120000 : (GENERATE)

MINUS CODE = 120001 : (GENERATE)

STAR CODE = 120002 : (GENERATE)

LESS CODE = 120014 : (GENERATE)

EQUAL CODE = 120220 : (GENERATE)

NOTUAL CODE = 120221 : (GENERATE)

GREATER CODE = 120017 : (GENERATE)

APNT : (NOTIFY)

MARK : (NOTIFY)

SLASH CODE = 121126 : (GENERATE)

COMMA CODE = 121025 : (GENERATE)

LEFT CODE = 120053 : (CLEAN)

RIGHT CODE = 124053 : (GENERATE)

LESSOR CODE = 122005 : (GENERATE)

CODE LINE FOR(C) AND : CHARS ' ' = : (FUNCTION)

CHARACTER VECTOR FOUND :

: (NOTIFY)

LPAREN CODE = 122002 : (GENERATE)

RPAREN CODE = 124002 : (GENERATE)

SEMICLN CODE = 122006

\*IF NEXT ELEMENT IS : OR : , INSECT ANULL :

CLEAN LINE FOR(C) ' ' = : (CLEAN)

LINE FOR(C) ANY(' ') : (GENERATE)

SAVED = RETURN

RETURN = 'MESSAGE' : (GENERATE)

MINDEX CODE = 120000

RETURN = 'MESSAGE' : (GENERATE)

MESSAGE2 RETURN = SAVED

CODE = 100000 : (GENERATE)

\*

IF CODE = 120002 : (GENERATE)

LIV CODE = 120003 : (GENERATE)

AND CODE = 120407 : (GENERATE)

OR CODE = 120410 : (GENERATE)

NOT CODE = 120411 : (GENERATE)

NAND CODE = 120412 : (GENERATE)

XOR CODE = 120413 : (GENERATE)

EE CODE = 120015 : (GENERATE)

GE CODE = 120016 : (GENERATE)

TOP CODE = 120004 : (GENERATE)

BOI CODE = 120005 : (GENERATE)

BAR CODE = 120006 : (GENERATE)

PI : (NOTIFY)

LOG : (NOTIFY)

DATA CODE = 121023 : (GENERATE)

RHO CODE = 121024 : (GENERATE)

ELMT : (NOTIFY)

TAKE : (NOTIFY)

DOUP : (NOTIFY)

ROT : (NOTIFY)

```

TRANS                : (NOT YET)
UP                   : (NOT YET)
DOWN                : (NOT YET)
DEC                 : (NOT YET)
ENC                 : (NOT YET)
EXPND  CODE = 121127 : (GENERATE)
O      CODE = 121032 : (GENERATE)
GOTO   CODE = 122000 : (GENERATE)
QUAD   CODE = 140000 : (GENERATE)
DOUAD  CODE = 144000 : (GENERATE)
*NEGATIVE #:
NEG     SIGN = 1
        LINE POS(0) FIGNUMBER * NUMBER = : F(ENRORD) S(NUMBER)
INNERPRD CODE = 121031 : (GENERATE)
LBRACE  DISPLACEMENT = 0
        DOUBLE = LINE
DISP    DISPLACEMENT = DISPLACEMENT + 2
        DOUBLE POS(0) (' ! SPAN(' ') NAME (' ! SPAN(' ') '3' = 2+
                : S(DISP)
        DOUBLE POS(0) (' ! SPAN(' ') NAME (' ! SPAN(' ') 2+
                'SBRACE' : F(ENRORD)
        CODE = 122001 : (GENERATE)
RBRACE CODE = 160000 + OCTAL(DISPLACEMENT) : (GENERATE)
*
*
```

```

* CREATE A NEW FUNCTION DEFINITION OR CLOSE THE PRESENT ONE
DEL      TOTALTOKENS = 0
        EC(FUNCTIONMODE,0)          :S(CNFUNCTION)
*CLOSE OLD FUNCTION:
        FUNCTIONMODE = 0
        LINENBR = LE(TOTALTOKENS,1) LINENBR - 1
*UPDATE # LOCAL VARIABLES (NEW LABELS) AND SHOW LABEL TABLE.
*CLEAR LOCAL PHANTOMS.
        OUT = '*LOCAL VARIABLES NUMBER IS ' (LPHANTOMNBR + 1)
        OUT = '*'
        OUT = '*****'
        OUT = '*THE LOCAL PHANTOM TABLE: '
        RESULT =
        N = -1
ERASELOCAL EC(LPHANTOMNBR,N)          :S(PRINTLABELS)
        N = N + 1
        OUT = '* ' N ' = ' (LPHCODE + OCTAL(N)) ' = ' S('LPHANTOM' N)
        S('LPHANTOMNBR' N) =          :S(ERASELOCAL)
*
*
PRINTLABELS OUT = '*****'
(      OUT = '*'
        OUT = '*LABELS NUMBER IS : ' (LABELNBR + 1)
        N = -1
ERASELABEL EC(LABELNBR,N)            :S(GFRZ)
        N = N + 1
        OUT = '* LABEL ' N ' IS ' S('LABEL' N)
        S('LABEL' N) =                :S(ERASELABEL)
*
*
*CREATE THE GFRZ ENTRY:
*****
GFRZ OUT = '*'
        FCNNBR = FCNNBR + 1
*CREATE 2ND WORD OF FC DESCRIPTOR
        PART3 = BIN(PARAMNBR)
MOREG PART3 = LT(SIZE(PART3),13) '0' PART3 :S(MOREG)
        PART1 = BIN(RR)
        PART2 = BIN(COUNT)
        PART2 = LT(SIZE(PART2),2) '0' PART2
        WORD2 = OCTBIN(PART1 PART2 PART3)
*LOAD GLOBAL TABLE:
        OUT = '*LOAD GFRZ WITH A NEW FUNCTION CALL TO ' FNAME ' : '
        OUT = ' CORE MADDR, ' (GLOBALBASE + 2 * FCNNBR) ': ' &+
        (FCCODE + OCTAL(FCNNBR)) 'B, ' WORD2 'B; '
*CREATE ENTRY IN FUNCTION TABLE:
        OUT = '*ENTRY IN FUNCTION TABLE: '
        ADDRESS = 200 + 4 * FCNNBR
        WORD1 = WORD2
        WORD2 = OCTAL(LPHANTOMNBR + 1)
        WORD3 = OCTAL(LIBASE + 1)
        WORD4 = OCTAL(LABELBASE)
        WORD4 = EC(LABELNBR,-1) W

```

```

OUT = '      CORE MADDR,' ADDRESS ':' ' WORD1 'B,' WORD2 'B,' &+
      WORD3 'E,' WORD4 'B;'
EQ(LABELNBR,-1)          :S(INPUTDONE)
LABELSTRING = UCIAL(LABELNBR + 1) 'B,' LABELSTRING
LABELSTRING ',' RPUS(0) = ';'
*CREATE LABEL TABLE:
OUT = '*CREATE LABEL TABLE : '
OUT = '      CORE MADDR,' LABELEASE ':' ' LABELSTRING : (INPUTDONE)
*
*
```

```

NOFUNCTION FUNCTIONMODE = 1
*COLLECT FUNCTION NAME,VARIABLES AND PARAMETERS:
    N = -1
    PARAMS =
    LIST =
    LPHANTOMNBR = -1
    PARAMFOUND = 0
    PARAMNBR = -1
    LABELSTRING =
    NEWOFLINES = 0
    LABELNBR = -1
    LINE POS(0) ARE . 11 'SLFRACE' ARE . PARAMS 'SEFRACE' &+
                                REM . 12      :F(DEFLOC)
    PARAMFOUND = 1
    LINE = 11 12
DEFLOC LINE BREAK(';') . HEAD =          :S(DOHEAD)
    HEAD = LINE
    LINE =
*THERE REMAINS ONLY 1 TO 3 NAMES AND EVENTUAL RESULT.
DOHEAD COUNT = 0
COUNTNAMES HEAD = TRIM(HEAD)
    HEAD NAME . S('NAME' COUNT) RPOS(0) =      :F(COUNTED)
    COUNT = COUNT + 1                          : (COUNTNAMES)
*
*COUNT NOW HAS # OF NAMES
* IF COUNT = 0 , NO ARG FCTION
*IF COUNT = 1 , 1 ARG FUNCTION
*IF COUNT = 2 , 2 ARG FUNCTION
COUNTED COUNT = COUNT - 1
    FNAME = EO(COUNT,0) S('NAME' 0) :S(PARAMENTER)
    FNAME = EO(COUNT,1) S('NAME' 1) :S(ARG2)
    FNAME = EO(COUNT,2) S('NAME' 1)  :S(ARG2) F(ERROR4)
*ENTER RIGHT ARG INTO LPHANTOMTABLE
ARG2 PHANTOM = S('NAME' 0)
    BACK = 'PARAMENTER'                          : (CREATELPH)
*
*PARAMS CONTAINS PARAM LIST IF ANY
PARAMENTER EO(PARAMFOUND,0)                      :S(LARG)
PARA      PARAMS POS(0) ' ' =                     :S(PARA)
    PARAMS = ';' PARAMS
    BACK = 'XTRACTPARAM'
XTRACTPARAM PARAMS = TRIM(PARAMS)
    PARAMNBR = PARAMNBR + 1
    PARAMS ';' NAME . PHANTOM RPOS(0) = :S(CREATELPH)
    IDEN1(PARAMS) :F(ERROR4)
*NO MORE PARAMS
*ENTER LEFT ARG IF ANY
LARG NE(COUNT,2)                                :S(RESEENTER)
*COUNT =2
    PHANTOM = S('NAME' 2)
    BACK = 'RESEENTER'                          : (CREATELPH)
*
CREATELPH LPHANTOMNBR = LPHANTOMNBR + 1

```

```

      $('LPHANTOM' LPHANTOMNR) = PHANTOM      : (CHECK)
      LPHANTOMNR = N                          : (BACK)
*
*
*ENTER RESULT IF ANY:
RESETER HEAD = TRIM(HEAD)
      HEAD ' ' =                               : (ISRESULT)
*NO RESULT
REMOV HEAD POS(0) ' ' =                       : (REMOVE)
      IDENT(HEAD,)                             : (GETLOCALS)
      NR = 0
*
*
ISRESULT HEAD = TRIM(HEAD)
      NR = 1
      HEAD NAME * RESULT RPOS(0)              : (ERROR4)
      PHANTOM = RESULT
      BACK = 'GETLOCALS'                      : (CREATELPH)
*LINE CONTAINS LOCALS IF ANY
GETLOCALS LINE = TRIM(LINE)
      IDENT(LINE,)                             : (LOCALSDONE)
      LINE ';'                                : (NOCLN)
*THERE IS A ;
      LINE ';' NAME * PHANTOM =               : (ERROR4)
      BACK = 'GETLOCALS'                     : (CREATELPH)
NOCLN IDENT(LINE,)                             : (ERROR4)
*ALL LOCAL PHANTOMS HAVE BEEN CREATED. CREATE FCN AND FCN TABLE
*THIS FCN IS A NEW GLOBAL NAME
LOCALSDONE PHANTOM = FNAME
      $('FUNCTION' FCNNR) = FNAME
      FCNNR = GPHANTOMNR + 1
      BACK = 'DOFTABLE'                       : (CREATEGPH)
DOFTABLE OUT = '* RESULT (<) = ' FNAME
      OUT = '* ARG# (AA) = ' COUNT
      PARAMNR = EG(PARAMNR,-1) 0
      OUT = '* PARAMETER # = ' PARAMNR
      OUT = '* LOCALPHANTOMS # = ' (LPHANTOMNR + 1)
      OUT = '*-----*'
      OUT = '* NEW FUNCTION : ' FNAME ' ! FUNCTION # = ' FCNNR
      OUT = '*-----*' : (END)
*

```

```

*
*
ERROR4 FUNCTIONMODE = 0
ERROR1 OUTPUT = 'SYNTAX ERROR : ' LINE
          LINENBR = GT(LINENBR,0) LINENBR - 1
          TOTALTOKENS = 0
          OUT = '*SYNTAX ERROR -- CANCELLED*'          :(AGAIN)
NOTYET OUTPUT = 'SORRY "" OPR "" IS NOT YET IMPLEMENTED.' :(AGAIN)
ERROR2 OUTPUT = 'MISSING QUOTE.TRY AGAIN.'          :(AGAIN)
SYSERROR OUTPUT = 'SYSTEM ERROR'          :(END)
*
*
*
GENERATE TOKENNR = TOKENNR + 1
          TOTALTOKENS = TOTALTOKENS + 1
          TOKENNR = GE(TOKENNR,5) 0          :F(NOCHANGE)
          STRING =          '21552152          ' STRING
NOCHANGE STRING = CODE 'B,' STRING          :(S(RETURNS))
EMIT CHAR('OUT')
*FILE IS NOW IN CHARACTER MODE
*DELETE INITIAL COMMA IN STRING
          STRING POS(0) ',' =
          OUT = ' CORE ADDR,' BASE ':' STRING ';'
          STRING =
**RE-ESTABLISH LINE MODE FOR FILE :
          LINE('OUT')
          OUT =
          NROFLINES = NROFLINES + 1
          LCOUNTER = LCOUNTER + 1
          LCOUNTER = EC(COUNTER,7) 0          :F(NOFEEED)
          LIST = LIST '21552152          '
NOFEEED LIST = LIST ',' BASE
ISOK          OUTPUT = 'OK'
          OUTPUT =          :(AGAIN)
*
*
INPUTDONE OUTPUT =
          OUT = '*THE LINE TABLE IS :'
          OUT = ' CORE ADDR,' LBASE ':' NROFLINES ',' *
          LIST ';'
          LIST =
          LBASE = LBASE + NROFLINES + 3
FCLOSED OUT = '*'
          LINENBR = 0
          LABELBASE = LABELBASE + (LABELNR + 1) * 3 :(ISOK)
*

```



```

ENDF IDENT(LINE,'END') :F(FULQSED)
N = -1
OUT = '*****'
OUT = '*THE GLOBAL PHANTOM TABLE : '
PRINTGPH EQ(GPHANTOMNR,N) :S(ASK)
N = N + 1
OUT = '* ' N ' = ' (GPHCODE + OCTAL(N)) ' = ' $('GPHANTOM' N)
$('GPHANTOM' N) = :C(PRINTGPH)
ASK OUTPUT =
OUT = '*****'
OUTPUT =
OUTPUT =
OUTPUTC = 'EXECUTION REQUESTED? '
GETL L = INPUTC
OUTPUT = IDENT(L,'Y') 'ES' :S(EXECUTE)
OUTPUT = IDENT(L,'N') 'O' :S(OUT)
OUTPUT = ' ? ' '... ' :C(GETL)
OUT OUTPUT = 'INTERNAL APL STRINGS READY ON :TEST*:*:'
CLOSE(-1)
Z = OPENIN(':TEST*:*:')
ASSIGN('GET11',Z)
OUTPUT =
TYPEFILE OUTPUT = GET11 :S(TYPEFILE)
CLOSE(-1) :C(END)
EXECUTE OUTPUT = 'CALLING THE SIMULATOR...'
OUTPUT = 'NOT YET IMPLEMENTED' :C(OUT)
OPEN A PSEUDO TELETYPE AND ASSEMBLE :

```

\*2 CASES: >0 AND <0 + 2 CASES: INTEGER AND FLOATING

\*FUNCTIONS:

```

FLOAT  X INTEGER . INT ('.' ! '') REM . MANT = INT MANT
        EXPONENT = SIZE(BIN(INT))
        EXPONENT = EC(X,0) 0
        Y        = BIN(EXPONENT)
        OUTPUT    = GT(SIZE(Y),7) 'EXPONENT OVERFLOW' :S(IN)
        BINMANT   = '0' BIN(X)
        EC(SIGN,1) :S(COMPLEMENT)
APPEND0 Y      = LT(SIZE(Y),8) '0' Y      :S(APPEND0)
        BINFLOAT = Y BINMANT
ADD0    BINFLOAT = LT(SIZE(BINFLOAT),32) BINFLOAT '0' :S(ADD0)
        BINFLOAT TAB(16) . LHALF REM . RHALF
        FLOAT    = OCTBIN(LHALF) 'B,' OCTBIN(RHALF) :S(RETURN)

```

\*

\*

```

COMPLEMENT Z = COMP(BINMANT)
            Z TAB(2) . BITS89 =
            BINMANT = IDENT(BITS89,'10') '10' Z      :S(APPEND0)
            BINMANT = IDENT(BITS89,'11') '1' Z '0'    :S(SYSEERR)

```

\*THE ABOVE IS A LEFT SHIFT BY ONE !

\*IT IS EASILY SHOWN THAT BIT 1 OF Z HAS TO BE 0.

```

        Y1 =
BINSUB1 Y LENC() . BIT RPOS(0) = :S(FIN2)
        Y1 = EC(BIT,1) '0' Y1      :S(FIN2)
        Y1 = EC(BIT,0) '1' Y1      :S(BINSUB1) S(SYSEERR)

```

\*

```

FIN2  Y = Y Y1      :S(APPEND0)

```

\*

\*

\*

\*

```

OCTAL  OCTAL =
OCT    OCTAL = LT(X,8) X OCTAL      :S(RETURN)
        QUOTIENT = X / 8
        REMAINDER = X - QUOTIENT * 8
        OCTAL = REMAINDER OCTAL
        X = QUOTIENT      :S(OCT)

```

\*

```

BIN    BIN = GT(X,1) BIN(X / 2) X - (X / 2) * 2 :S(RETURN)
        BIN = X      :S(RETURN)

```

\*COMPUT OCTAL FROM BINARY:

OCTBIN X = '00' X

\*(MAKES IT 18 IN LENGTH)

```

        OCTBIN =
OBL00P X LENC(3) . 1 RPOS(0) = :S(RETURN)
        OCTBIN = IDENT(1,'000') '0' OCTBIN :S(OBL00P)
        OCTBIN = IDENT(1,'001') '1' OCTBIN :S(OBL00P)
        OCTBIN = IDENT(1,'010') '2' OCTBIN :S(OBL00P)
        OCTBIN = IDENT(1,'011') '3' OCTBIN :S(OBL00P)
        OCTBIN = IDENT(1,'100') '4' OCTBIN :S(OBL00P)
        OCTBIN = IDENT(1,'101') '5' OCTBIN :S(OBL00P)
        OCTBIN = IDENT(1,'110') '6' OCTBIN :S(OBL00P)

```

```

      OCTBIN = IDENT(1,'111') '7' OCTBIN      :S(OBLLOOP) F(ERROR3)
*
*
*CONSTRUCT 2'S COMPLEMENT OF A NUMBER
COMP  COMP2 =
      BINMANT LEN(1) . BIT RPOS(0) = :F(BINPLUSONE)
      COMP = EQ(BIT,0) '1' COMP      :S(COMP)
      COMP = EQ(BIT,1) '0' COMP      :S(COMP) F(SYSEKERR)
*MUST ADD ONE IN BINARY:
BINPLUSONE COMP LEN(1) . BIT RPOS(0) =
      COMP2 = EQ(BIT,0) '1' COMP2      :S(FIN)
      COMP2 = EQ(BIT,1) '0' COMP2      :S(BINPLUSONE) F(SYSEKERR)
*
FIN  COMP = COMP COMP2      :S(RETURN)
*
```

\*THIS LISTS THE NORMALIZED VALUES OF ANY X.  
 \*IT IS SET TO LIST FROM -50 TO+50.  
 \*CAN BE EASILY CHANGED,NATURALLY FOR ANY RANGE.  
 \*

```

NORMALIZE OUTPUTC = 'MAX NUMBER = '
      MAX = INPUT
      OUTPUT =
      OUTPUT =
      OUTPUT =
      OUTPUT = 'DECIMAL :      O C T A L      :      B ' &-
      'I N A R Y'
      INTEGER = SPAN(6DIGITS)
      OUTPUT = '-----:-----:-----'
      '-----'
      DEFINE('OCTAL(X)')
      DEFINE('BIN(X)')
      DEFINE('OCTEIN(X)')
*
*X IS AN INTEGER TO BE FLOATED.NO LEADING ZEROS
      DEFINE('FLOAT(X)')
      DEFINE('COMF(X)')
      N = MAX
      SIGN = 1
NEGLOOP DECIMALN = X ' '
      DECIMALN = EC(SIGN,1) '-' DECIMALN
BLANKIT DECIMALN = LTCSIZE(DECIMALN,7) ' ' DECIMALN :S(BLANKIT)
      OCTALN = FLOAT(N) 'B'
*THIS COMPUTES ALSO LEAF AND RHAF
      BINARYN = BINFLOAT
*
      OUTPUT = DECIMALN ' : ' OCTALN ' : ' BINARYN
      EC(SIGN,0) :S(POSTEST)
      N = N - 1
      EC(N,0) :F(NEGLOOP)
POSLOOP SIGN = 0 :C(NEGLOOP)
*
POSTEST1 N = N + 1
      LE(N,MAX) :S(NEGLOOP)
      OUTPUT = :C(END)

```

## REPRESENTATION OF APL OPERATORS

## Appendix F - Representation of APL Operators

## 1 - Scalar Operators

APL symbol	Attributes	TTY	octal	Monadic	group	Dyadic	group
+	N	+	120000	identity	F	plus	F
-	N	-	120001	negation	F	minus	F
×	N	\$TIM	120002	signum	F	times	F
÷	N	\$DIV	120003	reciprocal	F	divide	F
*	N	*	120030	exponential	S	power	S
^	L	\$AND	120407	- -	F	and	F
∨	L	\$OR	120410	- -		or	F
~	L	\$NOT	120411	not		- -	
^	L	\$NAND	120412	- -		Nand	F
∨	L	\$NOR	120413	- -		Nor	F
<	N	<	120014	- -		less	F
≤	N	\$LE	120015	- -		less or equal	F
≥	N	\$GE	120016	- -		greater or equal	F
>	N	>	120017	- -		greater	F
=	N, C	=	120220	- -		equal	F
≠	N, C	#	120221	- -		not equal	F
⌈	N	\$TOP	120004	ceiling	F	Max	F
⌊	N	\$BOT	120005	floor	F	Min	F
	N	\$BAR	120006	magnitude	F	Residue	F
0	N	\$PI		pi-times	S	circ.	S
⊙	N	\$LOG		natural log	S	common log	S
!	N	!		factorial	S	combination	S
?	N	?		roll	S	deal	S

Attributes

N is Numeric  
L is Logical  
C is Character  
I is indexable

Group

F is firmware  
S is software

TTY

is Teletype representation

## 2 - Mixed Operators

APL symbol	Attributes	TTY	Octal	Monadic	group	Dyadic	group
ι		\$IOTA	121023	iota	F	index of	F
ρ		\$RHO	121024	size	F	reshape	F
,		,	121025	ravel	F	catenate	F
e		\$ELMT		- -		membership	S
↑		\$TAKE		- -		take	S
↓		\$DROP		- -		drop	S
⌈		\$UP		grade up	S	- -	
⌋		\$DOWN		grade down	S	- -	
⊥		\$DEC		- -		decode	S
⊤		\$ENC		- -		encode	S
⌹	I	\$ROT		reversal	S		
⌺	I	\$TRANS		mon. transpose	S	dyadic transpose	S
/	I	/	121126	reduction	F	compress	S
\	I	\	121127	- -		expand	S
}		}	122003	- -		indexing	F
[		[	124000	- -		indexing	F

## 3 - Others

APL symbol	Attributes	TTY	Octal	Monadic	group	Dyadic	group
○		\$o	121032	outer product	S	- -	
.		.	121031	inner product	S	- -	
←		←	122002	- -		assignment	F
→		\$GOTO	122000	branch	F	- -	
□		\$QUAD	140000	quad	F	- -	
▣		\$QQUAD	144000	quote-quad	F	- -	



## 4 - Other Symbols

APL symbol	TTY	Octal	name
'	'	- -	quote
(	(	122002	left paren
)	)	124002	right paren
{	\$LBRACE	122001	left brace
}	\$RBRACE	160000	right brace
∇	\$DEL	- -	delta
:	:	- -	colon
E	E	- -	exponent
;	;	124001	semicolon
-	\$NEG	- -	negative

## THE SOFTWARE APL OPERATORS

# Appendix G - Listings of the Software APL Operators

These APL functions have been written and tested by Ted Stohr, School of Business Administration. They have been limited to 2-dimensional arrays where applicable.

```

      ▽ Z←L COMPRESS R;LL;LR;LS;RR;V;I
[1]   →(2+TYPEL),Z←1J←0
[2]   →0,Z←ERROR 3+12×TYPEL
[3]   →4+ρJL←ρL
[4]   →(1-L=0)×7+12×L=1
[5]   →3+RR←ρLR←ρR
[6]   →0,Z←ERROR 11
[7]   →0,Z←ERROR 25
[8]   →4-2×1-1=LL
[9]   →2+(9×(LL=1)∧1-LR=1)+10×LL=LS←LR
[10]  →2+(9×(LL=1)∧1-LR[2]=1)+10×LL=LS←LR[2]
[11]  L←(LL←LS)ρL
[12]  →7+6×LL=+/(LLρ1)=(1=L)+0=L
[13]  →14×0<+/V←L×1LL
[14]  →15+(LL<I←I+1)×RR+1
[15]  →14+2×0<V[I]
[16]  →14,ρZ←Z,V[J]
[17]  →0,ρZ←P[Z]
[18]  →0,ρZ←R[;Z]
[19]  Z←R

```

▽

```

      ▽ Z←L COMPRESS1 R
[1]   Z←TRANPOSE L COMPRESS TRANPOSE R

```

▽

```

      ▽ Z←L COMPRESS2 R
[1]   Z←L COMPRESS R

```

▽

```

      V Z←L DROP R;L1;L2;R1;R2;LR;LL
[1]   →2+TYPEL
[2]   →0,ERROR 3+21×TYPEL
[3]   →(4+(ρLR←ρR)+3×ρLL←ρL)|10
[4]   →((1-1=|L)×2+(0=+/1|L)×9+L=0),Z←10
[5]   →2+(0=+/1|L)×9+3×((|L)<LR)∨LR=|L
[6]   →0,ERROR 12
[7]   →13-9×1=LL
[8]   →13-3×1=LL
[9]   →13+4×2=LL
[10]  →0,ERROR 11
[11]  →0,ERROR 31
[12]  →0,ρZ←R
[13]  →0,ERROR 14+13×(1=LL)∨2=LL
[14]  →((1-L=0)×(15×0<L)+16×L<0),ρZ←R
[15]  →0,ρZ←(,R)[L+1LR-L]
[16]  →0,ρZ←(,R)[1LR-|L]
[17]  →2+(0=+/1|L)×9+7×((L1<R1)∨(L1<|L[1])=(R1+LR[1]))^(L2<
      R2)∨(L2<|L[2])=R2+LR[2]
[18]  →19+(2×L[1]<0)+L[2]<0
[19]  →0,ρZ←R[(L1+1R1-L1);L2+1R2-L2]
[20]  →0,ρZ←R[(L1+1R1-L1);1R2-L2]
[21]  →0,ρZ←R[1R1-L1];L2+1R2-L2]
[22]  Z←R[(1R1-L1);1R2-L2]
      V

```

```

      ▽ Z←L DECODE R;LL;LR;RR;LS;W
[1]   →(2+((1-(0=TYPEL)∨0=TYPER)×(1-RR=2)×(1+RR+3×ρLL←ρL))+
      3×(1=TYPER)×2=RR←ρLR←ρR)| 9
[2]   →0,ERROR 3-2×0=TYPER
[3]   →0,Z←R
[4]   →(11×0<LR),(L←(LS←LR)ρL),Z←10
[5]   →0,ERROR 8
[6]   →(10+0<LL),R←(LS←LL)ρR
[7]   →((0<LR)×3-((2×LR=1)+4×LL=1)×1-LS←LR=LL),Z←10
[8]   →10+(0<LL)∧LL=LS←LR
[9]   →0,ERROR 11-3×2=ρLR
[10]  →0,ERROR 15-LL=0
[11]  W←((LL-1)ρI←0),1
[12]  →14-(I←J+1)<LS
[13]  →12,W[LS-I]←W[LS+1-I]×L[LS+1-I]
[14]  Z←+/W×R

```

▽

```

      V Z←L ENCODE R;I;LL;S;T
[1]   →2+(1-(0=TYPEH)∨0=TYPEP)×1+0<ppR
[2]   →0,ERROR 3-2×0=TYPEP
[3]   →5+ρLL←ρL
[4]   →0,ERROR 6+ppR
[5]   →6,(LL←1),L←,L
[6]   →(7+0<LL),(S←R),Z←LLρI←0
[7]   →0,ERROR 14-3×2=ρLL
[8]   →(9×(I←I+1)<1+LL),T←S
[9]   →10+0=L[LL+1-I]
[10]  S←[T:L[LL+1-I]
[11]  →(8×0<[L[LL+1-I]),Z[LL+1-I]←L[LL+1-I]|T
      V

```

```

      ▽ Z←L EXPAND R;RL;LL;RR;LR;LS;I;J;K
[1]   →(2+TYPEI×1+HYPER),K←ρJ←J←0
[2]   →0,ERROR 3
[3]   J←' '
[4]   →5+RL←ρLL←ρL
[5]   →6,(L←,L),LL←1
[6]   →11-(0←+/LL)×3-RR←ρLS←LR←ρR
[7]   →0,ERROR 11
[8]   →(12-LL<1),(ρZ←LLρJ),LS←LR←1
[9]   →(12-LL<LR),ρZ←LLρJ
[10]  →(12-LL<LS←LR[2]),ρZ←(LR[1],LL)ρJ
[11]  →0,Z←ERROR 15-0=+/LL
[12]  →13+LL=+/(LLρ1)=(1=L)+0=L
[13]  →0,Z←ERROR 25
[14]  →(15+(+/L)=LS),J←L×\LL
[15]  →0,Z←ERROR 31
[16]  →17+(LL<I←I+1)×2+RR=2
[17]  →18-2×0=J[I]
[18]  →16,K←K,J[I]
[19]  →0,ρZ[K]←R
[20]  Z[;K]←R

```

▽

```

      ▽ Z←GRADEUP X;L;LX;I;J;K;A;T
[1]   →(2+TYPEX×1+ρL←LX←ρX),A←Z←ρI←J←0
[2]   →0,ERROR 1
[3]   →0,ERROR 6
[4]   →(5×0<LX),(.X←(L+1,L)ρX),Z←10
[5]   →(6+4×L[1]<J←I+1),(A←1L[2]),(Z←Z,A),J←0
[6]   →7-2×(K←J←J+1)=L[2]
[7]   →3-2×L[2]<K←K+1
[8]   →7+2×X[I;K]<X[I;J]
[9]   →7,(A[K]←T),(A[J]←A[K]),(T←A[J]),(X[I;K]←T),(X[I;J]←X
      [I;K]),T←X[I;J]
[10]  Z←LXρZ
      ▽

```

```

      ▽ Z←GRADEDOWN X
[1]   Z←GRADEUP X×-1
      ▽

```



```

      ▽ Z←I, INDEX R;A;RL;LL;I;J;N
[1]   →2+1=RL←ρLL←ρL
[2]   →0,ERROR 9+RL
[3]   Z←(N←ρA←,P)ρI←0
[4]   →6-(N<I←I+1)+J←0
[5]   →0,,Z←(ρR)ρZ
[6]   →(7-3×LL<J),Z[I]←J←J+1
[7]   →6-2×(A[I]=L[J])
      ▽

```

```

      ▽ Z←L MEMBERSHIP R;L;N;A;B
[1]   →(2+0<RL←ρLL←ρL),(ρB←,R),Z←ρI←0
[2]   →0,Z←0<+/L=,R
[3]   H←ρA←,L
[4]   →5+N<J←I+1.
[5]   →4,Z←Z,0<+/A[I]=B
[6]   Z←LL,ρZ
      ▽

```

```

      V Z←L ROTATE R;RL;LL;RR;LR;I
[1]   →(2+TYPEL×1+1<PL←pLL←pL),I←0
[2]   →0,ERROR 3
[3]   →5+((0<+/LL)∨RL=0)×(0=+/1|L)×1+RP←pLR←pR
[4]   →0,ERROR 11
[5]   →0,ERROR 24-10×(0=+/LL)∧RL=1
[6]   →(9×1<p,L),pZ←P
[7]   →(((0<+/LR)∨1<p,L)×12+3×1=p,L),Z←10
[8]   →(10+1=p,L),pZ←LRpP[1;1]
[9]   →0,Z←ERROR 27
[10]  →12+LR[1]=LL
[11]  →13,I←LR[1]pL
[12]  →0,Z←ERROR 15+12×RR<2
[13]  →14×(I←I+1)<LR[1]+1
[14]  →13,pZ[I;]←P[I;1+LR[2]]~1+L[I]+1LR[2]]
[15]  Z←P[1+LR]~1+L+1LR]

```

V

```

      V Z←L TAKE R;L1;L2;R1;R2;LR;LL
[1]   →2+TYPEL
[2]   →0,ERROR 3+21×TYPEL
[3]   →(4+(ρLR←ρR)+3×ρLL+ρL)∣10
[4]   →((1-0=L)×2+(0=+/1∣L)×9+1=∣L),Z←10
[5]   →2+(0=+/1∣L)×9+3×((∣L) < LR)∨LR=∣L
[6]   →0,ERROR 12
[7]   →13-9×1=LL
[8]   →13-8×1=LL
[9]   →13+4×2=ρL
[10]  →0,ERROR 11
[11]  →0,ERROR 31
[12]  →0,ρZ←R
[13]  →0,ERROR 14+13×(1=LL)∨2=LL
[14]  →((1-L=0)×(15×0<L)+16×L<0),Z←10
[15]  →0,ρZ←(,R)[1L]
[16]  →0,ρZ←(,R)[(LR-∣L)+1∣L]
[17]  →2+(0=+/1∣L)×9+7×((L1<R1)∨(L1←∣L[1])=(R1+LR[1]))^(L2<
      R2)∨(L2←∣L[2])=R2←LR[2]
[18]  →19+(2×L[1]<0)+L[2]<0
[19]  →0,ρZ←R[(1L1);1L2]
[20]  →0,ρZ←R[(1L1);(P2-L2)∖>L2]
[21]  →0,ρZ←P[((F1-L1)+1L1);1L2]
[22]  Z←R[((P1-L1)+1L1);(R2-L2)+1L2]

```

∇

```

      ▽ Z←TRANSPOSE X;LX;I
[1]   →(2+2=ρLX←ρX),I←0
[2]   →0,ρZ←X
[3]   Z←(LX[2],LX[1])ρX[1;1]
[4]   →5×(I←I+1)<1+LX[1]
[5]   →4,ρZ[(\LX[2]);I]←X[I;\LX[2]]
      ▽

```

▽ Z←ERROR N

```

[1] →(N+1),Z←10
[2] →0,p[←'1 - TYPE ERROR - ILLEGAL LITERAL RIGHT-HAND ARGUMENT'
[3] →0,p[←'2 - TYPE ERROR - ILLEGAL NUMERIC RIGHT-HAND ARGUMENT'
[4] →0,p[←'3 - TYPE ERROR - ILLEGAL LITERAL LEFT-HAND ARGUMENT'
[5] →0,p[←'4 - TYPE ERROR - ILLEGAL NUMERIC LEFT-HAND ARGUMENT'
[6] →0,p[←'5 - TYPE ERROR - ILLEGAL MIXED ARGUMENT TYPES'
[7] →0,p[←'6 - RANK ERROR - ILLEGAL SCALAR RIGHT-HAND ARGUMENT'
[8] →0,p[←'7 - RANK ERROR - ILLEGAL VECTOR RIGHT-HAND ARGUMENT'
[9] →0,p[←'8 - RANK ERROR - ILLEGAL MATRIX RIGHT-HAND ARGUMENT'
[10] →0,p[←'9 - RANK ERROR - ILLEGAL SCALAR LEFT-HAND ARGUMENT'
[11] →0,p[←'10 - RANK ERROR - ILLEGAL VECTOR LEFT-HAND ARGUMENT'
[12] →0,p[←'11 - RANK ERROR - ILLEGAL MATRIX LEFT-HAND ARGUMENT'
[13] →0,p[←'12 - RANK ERROR - ARGUMENT RANKS NOT CONFORMABLE'
[14] →0,p[←'13 - LENGTH ERROR - ILLEGAL RIGHT-HAND ARGUMENT LENGTH'
[15] →0,p[←'14 - LENGTH ERROR - ILLEGAL LEFT-HAND ARGUMENT LENGTH'
[16] →0,p[←'15 - LENGTH ERROR - ARGUMENT LENGTHS NOT CONFORMABLE'
[17] →0,p[←'16 - VALUE ERROR - ILLEGAL RIGHT-HAND ARGUMENT VALUE'
[18] →0,p[←'17 - VALUE ERROR - ILLEGAL ZERO RIGHT-HAND ARGUMENT VALUE'
[19] →0,p[←'18 - VALUE ERROR - ILLEGAL NEGATIVE RIGHT-HAND ARGUMENT VALUE'
[20] →0,p[←'19 - VALUE ERROR - ILLEGAL NON-INTEGGER RIGHT-HAND ARGUMENT VALUE'
[21] →0,p[←'20 - VALUE ERROR - ILLEGAL NON-LOGICAL RIGHT-HAND ARGUMENT VALUE'
[22] →0,p[←'21 - VALUE ERROR - ILLEGAL LEFT-HAND ARGUMENT VALUE'
[23] →0,p[←'22 - VALUE ERROR - ILLEGAL ZERO LEFT-HAND ARGUMENT VALUE'
[24] →0,p[←'23 - VALUE ERROR - ILLEGAL NEGATIVE LEFT-HAND ARGUMENT VALUE'
[25] →0,p[←'24 - VALUE ERROR - ILLEGAL NON-INTEGGER LEFT-HAND ARGUMENT VALUE'
[26] →0,p[←'25 - VALUE ERROR - ILLEGAL NON-LOGICAL LEFT-HAND ARGUMENT VALUE'
[27] →0,p[←'26 - RANK/LENGTH ERROR'
[28] →0,p[←'27 - LENGTH/RANK ERROR'
[29] →0,p[←'28 - RANK/VALUE ERROR'
[30] →0,p[←'29 - VALUE/RANK ERROR'
[31] →0,p[←'30 - LENGTH/VALUE ERROR'
[32] '31 - VALUE/LENGTH ERROR'

```

▽

## TABLE OF FLOATING POINT INTEGERS

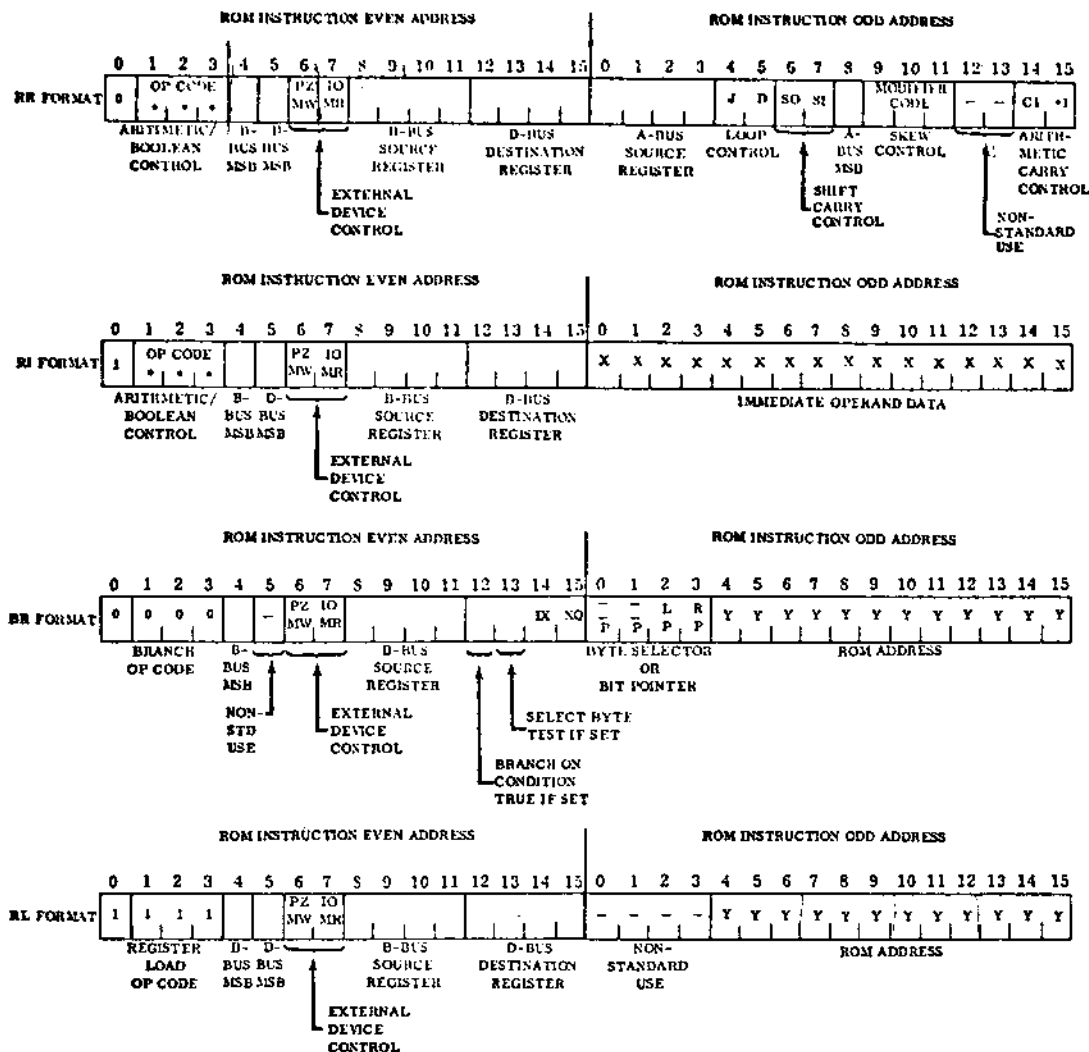
## Appendix H - Table of Floating-point Integers

DECIMAL	:	O C T A L	:	B I N A R Y
-25	:	002634B,000000B	:	00000101100111000000000000000000
-24	:	002640B,000000B	:	00000101101000000000000000000000
-23	:	002644B,000000B	:	00000101101001000000000000000000
-22	:	002650B,000000B	:	00000101101010000000000000000000
-21	:	002654B,000000B	:	00000101101011000000000000000000
-20	:	002660B,000000B	:	00000101101100000000000000000000
-19	:	002664B,000000B	:	00000101101101000000000000000000
-18	:	002670B,000000B	:	00000101101110000000000000000000
-17	:	002674B,000000B	:	00000101101111000000000000000000
-16	:	002200B,000000B	:	00000100100000000000000000000000
-15	:	002210B,000000B	:	00000100100010000000000000000000
-14	:	002220B,000000B	:	00000100100100000000000000000000
-13	:	002230B,000000B	:	00000100100110000000000000000000
-12	:	002240B,000000B	:	00000100101000000000000000000000
-11	:	002250B,000000B	:	00000100101010000000000000000000
-10	:	002260B,000000B	:	00000100101100000000000000000000
-9	:	002270B,000000B	:	00000100101110000000000000000000
-8	:	001600B,000000B	:	00000011100000000000000000000000
-7	:	001620B,000000B	:	00000011100100000000000000000000
-6	:	001640B,000000B	:	00000011101000000000000000000000
-5	:	001660B,000000B	:	00000011101100000000000000000000
-4	:	001200B,000000B	:	00000010100000000000000000000000
-3	:	001240B,000000B	:	00000010101000000000000000000000
-2	:	000600B,000000B	:	00000000110000000000000000000000
-1	:	000200B,000000B	:	00000000100000000000000000000000
0	:	000000B,000000B	:	00000000000000000000000000000000
1	:	000500B,000000B	:	00000000101000000000000000000000
2	:	001100B,000000B	:	00000010010000000000000000000000
3	:	001140B,000000B	:	00000010011000000000000000000000
4	:	001500B,000000B	:	00000011010000000000000000000000
5	:	001520B,000000B	:	00000011010100000000000000000000
6	:	001540B,000000B	:	00000011011000000000000000000000
7	:	001560B,000000B	:	00000011011100000000000000000000
8	:	002100B,000000B	:	00000100010000000000000000000000
9	:	002110B,000000B	:	00000100010010000000000000000000
10	:	002120B,000000B	:	00000100010100000000000000000000
11	:	002130B,000000B	:	00000100010110000000000000000000
12	:	002140B,000000B	:	00000100011000000000000000000000
13	:	002150B,000000B	:	00000100011010000000000000000000
14	:	002160B,000000B	:	00000100011100000000000000000000
15	:	002170B,000000B	:	00000100011110000000000000000000
16	:	002500B,000000B	:	00000101010000000000000000000000
17	:	002504B,000000B	:	00000101010001000000000000000000
18	:	002510B,000000B	:	00000101010010000000000000000000
19	:	002514B,000000B	:	00000101010011000000000000000000
20	:	002520B,000000B	:	00000101010100000000000000000000
21	:	002524B,000000B	:	00000101010101000000000000000000
22	:	002530B,000000B	:	00000101010110000000000000000000
23	:	002534B,000000B	:	00000101010111000000000000000000
24	:	002540B,000000B	:	00000101011000000000000000000000
25	:	002544B,000000B	:	00000101011001000000000000000000





# META4 COMPUTER CONTROL INSTRUCTIONS



#### ARITHMETIC/BOOLEAN CONTROL OP CODES

MNEMONIC	BINARY VALUE
BRZ BNZ	000
AND	001
OR	010
XOR	011
ADD	100
MULT	101
DIV	110
LOAD	111

#### SKW CONTROL MODIFIER CODES

MNEMONIC	BINARY VALUE
(NONE)	000
R1	001
L1	010
SK	011
SE	100
R8	101
L8	110
EX	111

#### MISCELLANEOUS MODIFIERS

SYMBOL	SYMBOL
C1	L
D	PZ
IO	R
IX	SI
J	SO
MR	XQ
MW	+1
W	

#### ARITHMETIC/BOOLEAN CONTROL MICROASSEMBLER PSEUDO-OP CODES

COPY	HO
JMP	HNO
NOP	HS
BRP	BNS
BRN	LDI
BC	SUM
BNC	

#### ASSEMBLER CONTROL PSEUDO-OP CODES

EQU
EQR
ORG
END
LIST
PNCH
EJCT

#### DATA STATEMENT PSEUDO-OP CODE

HEX

BNZ

Format RR     BRANCH IF NONZERO CONDITION

Valid Modifiers: R L R, L W IX XQ PZ IO

The condition specified by modifiers or a register bit position is tested. If the test result is nonzero, the next instruction is taken from the even ROM location specified in the operand field. If the test result is zero, the next sequential instruction is executed. Registers and machine conditions are not changed. The operand field must contain either a label or an absolute address. The least significant bit of an address is ignored and interpreted as zero. Logical indexing applies if IX modifier is specified.

BRZ

Format BR     BRANCH IF ZERO CONDITION

Valid Modifiers: R L R, L W IX XQ PZ IO

The condition specified by modifiers or by a bit position is tested. If the test result is zero, the next instruction is taken from the ROM location specified in the operand field. If the result is nonzero, the next sequential instruction is executed. Registers and machine conditions are not changed. The operand field must contain either a label or an absolute address. The least significant bit of an address is ignored and interpreted as zero. Logical indexing applies if IX modifier is specified. The use of mnemonics BNZ and BRZ is shown in the following table:

	Bits 0 - 7 zero Bits 8 - 15 zero	Bits 0 - 7 zero Bits 8 - 15 nonzero	Bits 0 - 7 nonzero Bits 8 - 15 zero	Bits 0 - 7 nonzero Bits 8 - 15 nonzero
BRZ W	•			
BRZ R	•		•	
BRZ L	•	•		
BRZ R, L	•	•	•	
BNZ R, L				•
BNZ R		•		•
BNZ L			•	•
BNZ W		•	•	•

• indicates conditions for successful branch

NOTE: Branch testing of an I/O register input must not be attempted unless the I/O system is stabilized at the time. Stabilization is assured by input/output system data via timing interlocks. Stabilization is not assured for non-synchronized inputs such as those used for interrupts. The effect of testing a nonstabilized input may be a ROM program branch to an address which is neither the next sequential address nor the expected branch address.

## AND

Format RB LOGICAL AND

Valid Modifiers: R1 L1 SK SE R8 L8 EX SI  
SO J D PZ IO MR MW

The contents of the A-register and the B-register are AND'ed bit by bit. The result is stored in the D-register. The four possible AND'ing results are:

BIT VALUES

A-REGISTER	1	1	0	0
B-REGISTER	1	0	1	0
D-REGISTER RESULT	1	0	0	0

The contents of the A- and B-registers are left unchanged by this operation.

## OR

Format RR LOGICAL INCLUSIVE OR

Valid Modifiers: R1 L1 SK SE R8 L8 EX SI  
SO J D PZ IO MR MW

The contents of the A-register and the B-register are inclusive OR'ed bit by bit. The result is stored in the D-register. The four possible OR'ing results are:

BIT VALUES

A-REGISTER	1	1	0	0
B-REGISTER	1	0	1	0
D-REGISTER RESULT	1	1	1	0

The contents of the A- and B-registers are left unchanged by this operation.

## XOR

Format RR LOGICAL EXCLUSIVE OR

Valid Modifiers: R1 L1 SK SE R8 L8 EX SI  
SO J D PZ IO MR MW

The contents of the A-register and the B-register are Exclusive OR'ed bit by bit. The result is stored in the D-register. The four possible XOR'ing results are:

BIT VALUES

A-REGISTER	1	1	0	0
B-REGISTER	1	0	1	0
D-REGISTER RESULT	0	1	1	0

The contents of the A- and B-registers are left unchanged by this operation.

## ADD

Format RR ADD

Valid Modifiers: R1 L1 SK SE R8 L8 EX SI  
SO J D PZ IO MR MW CI +1

The contents of the A-register and the B-register are added. The result is stored in the D-register. Addition is carried out in two's complement format. Carry input to the least significant bit is controlled by CI, which enables the previous carry condition as input; and by +1, which forces carry input. The carry condition is set to correspond to the carry from bit 0 and the overflow is set to correspond to the Exclusive OR of the carries from bits 1 and 0. The A- and B-registers are left unchanged by this operation, but the carry and overflow bits are changed.

LABEL	OPERATION	B REG	D REG	A REG	OPERAND
TAG	ADD	2	14	3	

In this example, the contents of Registers 2 and 3 are added and the sum appears in Register 14.

## MULT

## Format RR    MULTIPLY STEP

Valid Modifiers: R1 L1 SK SE R8 L8 EX SI  
                   SO J D PZ IO MR MW CI +1

If the shift condition is 1 prior to the MULT instruction, the contents of the A- and B-registers are added. The result is stored in the D-register. Addition is carried out in two's complement format. Carry input to the least significant bit is controlled by CI, which enables the previous carry condition as input; and by +1, which forces carry input. The carry condition is set to correspond to the carry from bit 0 and the overflow is set to correspond to the Exclusive OR of the carries from bits 1 and 0. The A- and B-registers are left unchanged by this operation, but the carry and overflow bits are changed.

If the shift condition is zero prior to the MULT instruction execution, the B-register data is inhibited so that the A-register data passes through the input adder unchanged and is stored in the D-register.

## DIV

## Format RR    DIVIDE STEP

Valid Modifiers: R1 L1 SK SE R8 L8 EX SI  
                   SO J D PZ IO MR MW CI +1

The contents of the A- and B-registers are added. If the sum is positive, the result is stored in the D-register. If the sum is a negative number, the D-register is not changed. Addition is carried out in two's complement format. Carry input to the least significant bit is controlled by CI, which enables the previous carry condition as input; and +1, which forces carry input. The carry condition is set to correspond to the Exclusive OR of the carries of bits 1 and 0. The carry and overflow conditions, and the shift condition (if SO is specified) are changed by the DIV instruction whether the sum is positive or negative.

**ANDI****Format RI LOGICAL AND IMMEDIATE**

Valid Modifiers: PZ IO MR MW

The contents of the B-register and the operand field are AND'ed bit by bit. The result is stored in the D-register. The operand must be either a left-justified hexadecimal constant or a label.

IMMEDIATE OPERAND	1	1	0	0
B-REGISTER	1	0	1	0
D-REGISTER RESULT	1	0	0	0

The contents of the B-register are left unchanged by this instruction.

**ORI****Format RI LOGICAL INCLUSIVE OR IMMEDIATE**

Valid Modifiers: PZ IO MR MW

The contents of the B-register and the operand field are Inclusive OR'ed bit by bit. The result is stored in the D-register. The contents of the operand field must be either a left-justified hexadecimal constant or a label.

IMMEDIATE OPERAND	1	1	0	0
B-REGISTER	1	0	1	0
D-REGISTER RESULT	1	1	0	0

The contents of the B-register are left unchanged by this instruction.

**XORI****Format RI LOGICAL EXCLUSIVE OR IMMEDIATE**

Valid Modifiers: PZ IO MR MW

The contents of the B-register and the operand field are Exclusive OR'ed bit by bit. The result is stored in the D-register. The operand field must be either a left-justified hexadecimal constant or a label.

IMMEDIATE OPERAND	1	1	0	0
B-REGISTER	1	0	1	0
D-REGISTER	0	1	1	0

The contents of the B-register are left unchanged by this instruction.

**ADDI**

Format RI      LOGICAL ADD IMMEDIATE

Valid Modifiers: PZ IO MR MW

The contents of the B-register and the operand field are added. The sum is stored in the D-register. The operand must contain a left-justified hexadecimal constant, or a label. Addition is carried out in two's complement format. The carry condition is set to correspond to the carry from bit 0 and the overflow is set to correspond to the Exclusive OR of the carries from bits 1 and 0.

The B-register is left unchanged by this instruction.

**LOAD**

Format RL      LOAD REGISTER

Valid Modifiers: PZ IO MR MW

The contents of the B-register and a ROM cell are Exclusive OR'ed bit by bit. The result is stored in the D-register. The ROM cell is specified by the contents of the Link register Inclusive OR'ed with the operand field. The operand field must contain either a left-justified hexadecimal constant or a label. Effective addresses may be even or odd.

The contents of the B-register are unchanged by this instruction.

**ROM**  
**Instruction**  
**Modifiers**

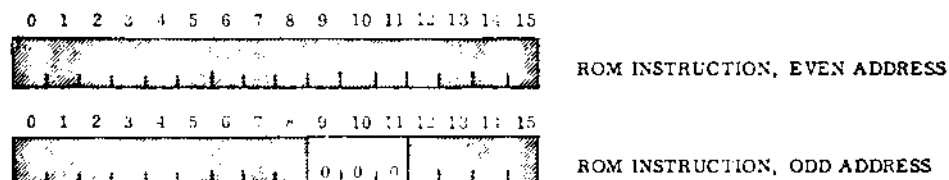
•The ROM instruction modifiers are described in detail on the following pages. They are grouped as outlined below:

- Skew Control Modifiers
- Arithmetic Control Modifiers
- Instruction Loop Repeat Control Modifiers
- Branch Control Modifiers
- Input/Output and Memory Control Modifiers



(No Skew  
Control  
Modifier)

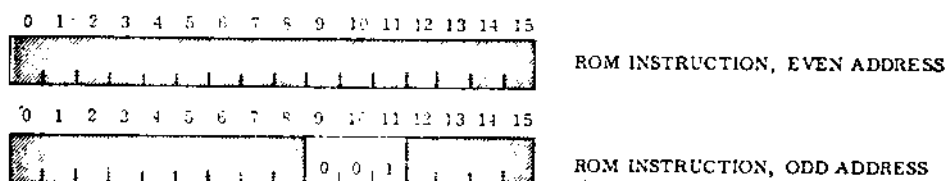
## TRANSMIT DATA WITHOUT MODIFICATION



Output from the Arithmetic unit is transmitted without modification.

R1

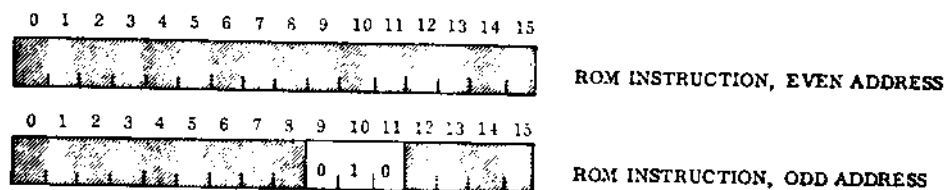
## SHIFT RIGHT ONE PLACE



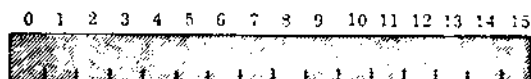
Output from the Arithmetic unit is displaced right one place. Spill from bit 15 may be saved in the Shift Condition register bit by the SO modifier. Entry to bit 0 from the previous shift condition is controlled by the SI modifier.

L1

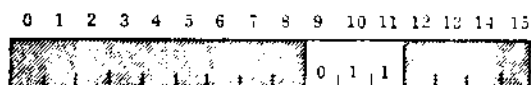
## SHIFT LEFT ONE PLACE



Output from the Arithmetic unit is displaced left one place. Spill from bit 0 may be saved in the Shift Condition register bit by the SO modifier. Entry into bit 15 from the previous shift condition is controlled by the SI modifier.



ROM INSTRUCTION, EVEN ADDRESS



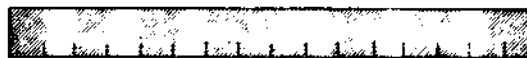
ROM INSTRUCTION, ODD ADDRESS

MULT  
DIV  
←

Output from the Arithmetic unit is displaced one place to the right. Spill from bit 15 may be saved by the SO modifier. Entry to bit 0 is made from the current arithmetic carry during an ADD operation or from the Carry Condition register bit during operations other than ADD. If the SI modifier is specified concurrently with SK, entry to bit 15 is the OR between the Shift Condition register bit and the proper carry condition.

## SIGN EXTEND

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15



ROM INSTRUCTION, EVEN ADDRESS

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15



ROM INSTRUCTION, ODD ADDRESS

Output from the Arithmetic unit is transmitted with bits 0 through 7 replaced by copies of bit position 8.

## SHIFT RIGHT EIGHT PLACES

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15



ROM INSTRUCTION, EVEN ADDRESS

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15



ROM INSTRUCTION, ODD ADDRESS

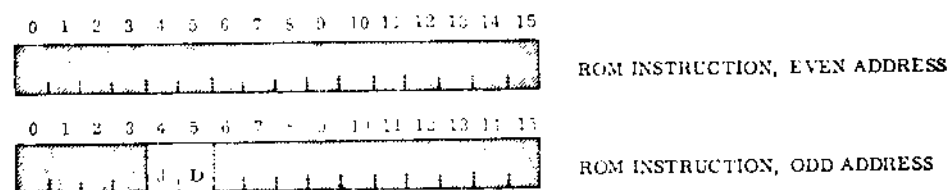
Output from the Arithmetic unit is displaced right eight places. Spill from the right is lost; zeros enter at the left.

D

## DECREMENT COUNTER

J

## JUMP ON COUNTER NONZERO



The low-order 8 bits of the Loop Counter are decremented and tested using the J and D modifiers. If J is specified without D, a branch to the address specified by the contents of the Link register occurs.

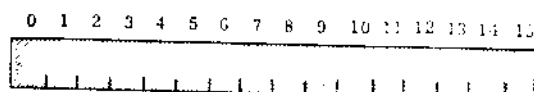
If D is specified without J, the counter is decremented.

If J is specified concurrently with D, a branch to the address specified by the contents of the Link register occurs unless the counter decrements to zero. The test is made after conclusion of the instruction.

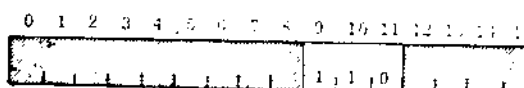
L8

## SHIFT LEFT EIGHT PLACES

Output from the Arithmetic unit is displaced left eight places. Spill from the left is lost; zeros enter at the right.



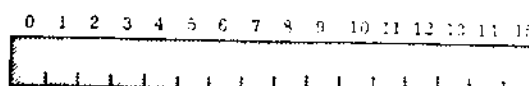
ROM INSTRUCTION, EVEN ADDRESS



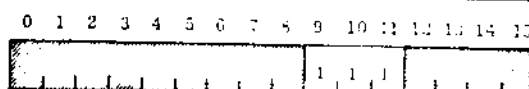
ROM INSTRUCTION, ODD ADDRESS

EX

## EXCHANGE BYTES



ROM INSTRUCTION, EVEN ADDRESS



ROM INSTRUCTION, ODD ADDRESS

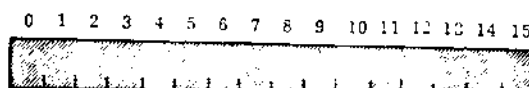
Output from the Arithmetic unit is rotated eight places so that bits 0 through 7 and bits 8 through 15 are interchanged.

CI

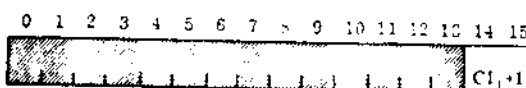
## ENABLE ARITHMETIC CARRY INPUT

+1

## FORCE ARITHMETIC CARRY INPUT



ROM INSTRUCTION, EVEN ADDRESS



ROM INSTRUCTION, ODD ADDRESS

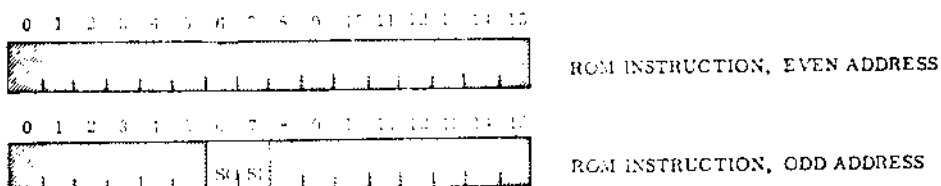
Carry input to the adder is controlled by CI and +1 modifiers. If neither is specified, the add is without carry input. If CI is specified, the previous carry condition is used as carry input to the least significant stage of the adder. If +1 is specified, a carry input is forced unconditionally, regardless of whether CI is also specified.

SO

SHIFTER OUTPUT SPILL TO SHIFT CARRY BIT

SI

SHIFTER INPUT ENTRY FROM SHIFT CARRY BIT



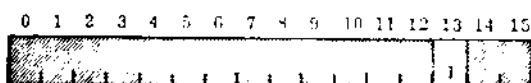
Shifter spill is always from either bit 0 or bit 15 of the operand being shifted. SO is effective for any shifter control modifier code. Spill is the original contents of bit 0 for no shift, L1, SE, and L8; and the original contents of bit 15 for R1, SK, R8, and EX. Refer to the DIV instruction description for use of the SO modifier with operations other than a shift.

Shifter input is taken from the Shift Condition register bit when the SI modifier is specified. The SI modifier is enabled only for R1, L1, and SK modifiers and controls either bit 0 or bit 15 entry, as appropriate. If SK and SI modifiers are specified concurrently, the entry to bit 0 is the OR between the shift condition and the arithmetic carry.

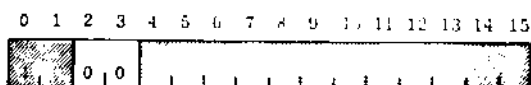
Circular shifts (end around) may be implemented by first executing a single shift operation (right or left, as appropriate) with register zero as the destination and SO specified. The shift carry bit will then be properly set so that subsequent shift operations with both SO and SI specified will be a circular shift.

W

## TEST WORD (RIGHT AND LEFT BYTES)



ROM INSTRUCTION, EVEN ADDRESS

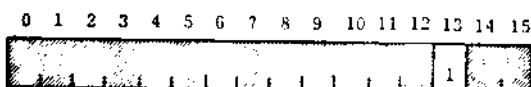


ROM INSTRUCTION, ODD ADDRESS

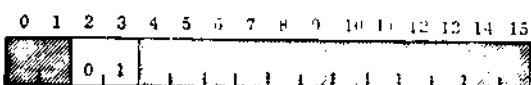
The contents of the register referenced by the B-bus address is tested for zero or nonzero condition.

R

## RIGHT BYTE TEST



ROM INSTRUCTION, EVEN ADDRESS

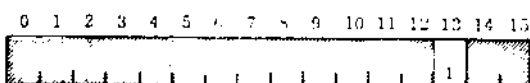


ROM INSTRUCTION, ODD ADDRESS

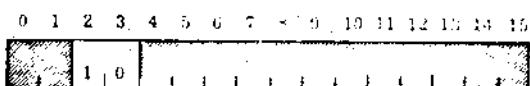
The right byte (8 bits) of the contents of the register referenced by the B-bus address is tested for zero or nonzero condition.

L

## LEFT BYTE TEST



ROM INSTRUCTION, EVEN ADDRESS

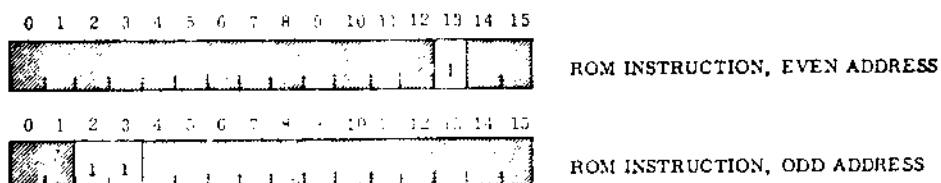


ROM INSTRUCTION, ODD ADDRESS

The left byte (8 bits) of the contents of the register referenced by the B-bus address is tested for zero or nonzero condition.

R, L

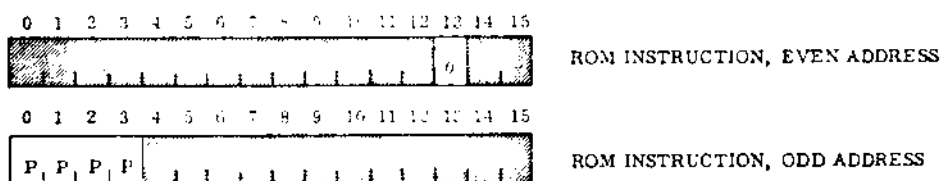
## RIGHT OR LEFT BYTE TEST



The right byte (8 bits) and left byte (8 bits) of the contents of the register referenced by the B-bus address are checked independently for zero or nonzero, with the Inclusive OR of the results tested for the zero or nonzero condition.

(No Byte  
Test  
Modifiers)

## TEST SPECIFIED BIT



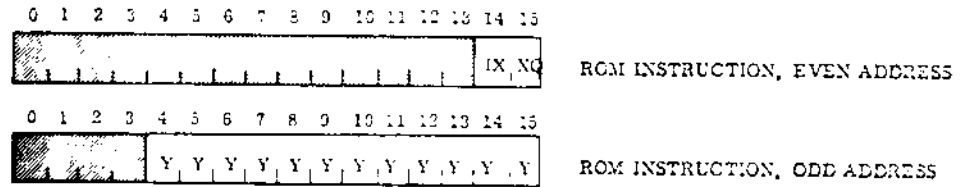
One bit of the contents of the register referenced by the B-bus address and the P-field is tested for zero or nonzero. This modifier enables a test and branch capability on any bit of any register and encompasses tests for even/odd, positive/negative, arithmetic carry, arithmetic overflow, and shift carry. The 4-bit P-field (pointer) is decoded to define one of 16 bit positions within the tested word. Pseudo-operation mnemonics are defined for positive/negative, carry, overflow, and shift condition tests.

IX

## LOGICAL INDEX

XQ

## EXECUTE ONE INSTRUCTION AFTER BRANCH



Logical indexing, if selected, OR's the contents of the Link register with the Y-field to form the effective address.

Execute mode inhibits changing the ROM address register if the Branch instruction test is successful. The effective address is used directly for the execution of one instruction and the control sequence then reverts back to that instruction which would have been executed had the Branch not been successful, unless that one instruction is itself a Branch instruction. Multilevel Branch and Execute instructions may be used with ultimate reversion of control back to that instruction which would have been done with only one level of Branch and Execute. If a Branch without Execute is in the multilevel Branch sequence, then reversion of control will not occur if the Branch without Execute is successful.

Execute mode may be considered as a capability for calling a one-instruction subroutine. The address in ROM is taken directly from the Link register and Y-field of the instruction, as appropriate, but the ROM address register is inhibited from copying the out-of-sequence ROM address reference.

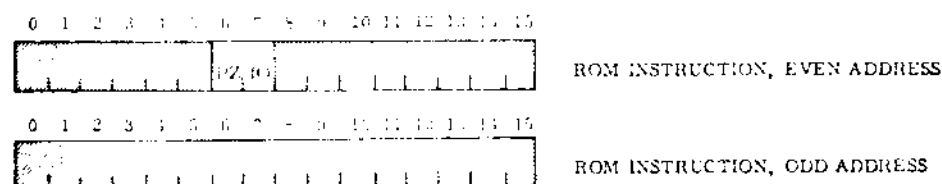


IO

## OUTPUT I/O REGISTER CONTROL SIGNALS

PZ

## PROGRAM PAUSE FOR CONTROL SIGNAL INPUT



PZ and IO functions are enabled when any I/O register is specified by the B-bus or D-bus address fields of the instruction word. When the PZ modifier is specified, the program pauses until a control flip-flop is cleared by an external signal and the clearing pulse has terminated. If more than one I/O register is specified by the bus register address fields of the instruction word, the pause condition occurs while any one of the associated control flip-flops is set and the control signal is output on all of the control lines. When the IO modifier is specified, the I/O register control line signals are output and control flip-flops are set. The control flip-flops are cleared by the external equipment or by the computer Master Clear.

Pause occurs prior to any instruction execution; I/O control signals are output at the conclusion of the instruction execution. Concurrent PZ and IO selection operates to delay the instruction until the control flip-flop is reset (pause condition is released); the instruction then executes, and the IO control flip-flop is set again.

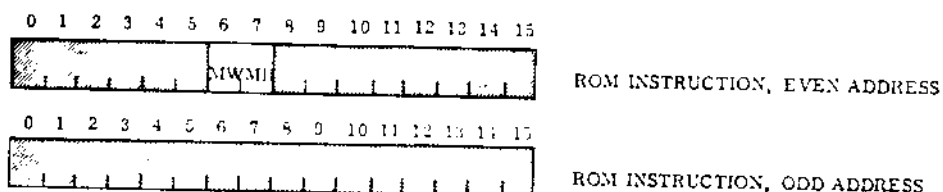
Note the possible conflicts: MW and MR modifiers use the same control word bit positions as PZ and IO modifiers. Conflicts may occur if memory and I/O registers are specified concurrently.

MW

## INITIATE MEMORY WRITE

MR

## INITIATE MEMORY READ



MW and MR modifiers are recognized only when a core memory register is specified by the D-bus register address field. When either MW or MR is specified, control flip-flops are set. The control flip-flops are cleared by signals from the memory. While the control flip-flops are set, the program will pause if the register address specified in the next instruction is either a B-bus or a D-bus register address and will resume when the memory has completed a cycle.

Note the possible conflicts: When MR and MW are specified simultaneously, the core memory will change protect bits to the state specified by bit 15 of the memory data register. Data will be restored without change. PZ and IO modifiers use the same control word bit positions as MW and MR modifiers. Conflicts may occur if memory and I/O registers are addressed concurrently.

A restriction must be observed when reading data from the Memory Data register subsequent to initiation of a Read operation. Memory Read data is valid only if read during the approximately 8 ROM instruction cycles following initiation of the Memory Read cycle, unless a buffered memory register is used, and a PZ bit must be used with memory register address in the B-bus field unless:

1. The data has been previously read using a PZ bit and cannot have changed since, or
2. A memory register is also specified in the D-bus field with either MR or MW specified.

## THE HARDWARE MAP

## Appendix J - The Hardware Map

The APL machine's physical memory is organized in 512 word pages and managed through a hardware address translator, the map.

Each process' virtual space is 64K long and may thus reference 128 pages. The initial system configuration can use a 64K main memory, but the map allows for future expansion to 256K. The hardware map is 128 words long, 12 bits wide. It completely defines the virtual memory accessible to any one process (see figure J1).

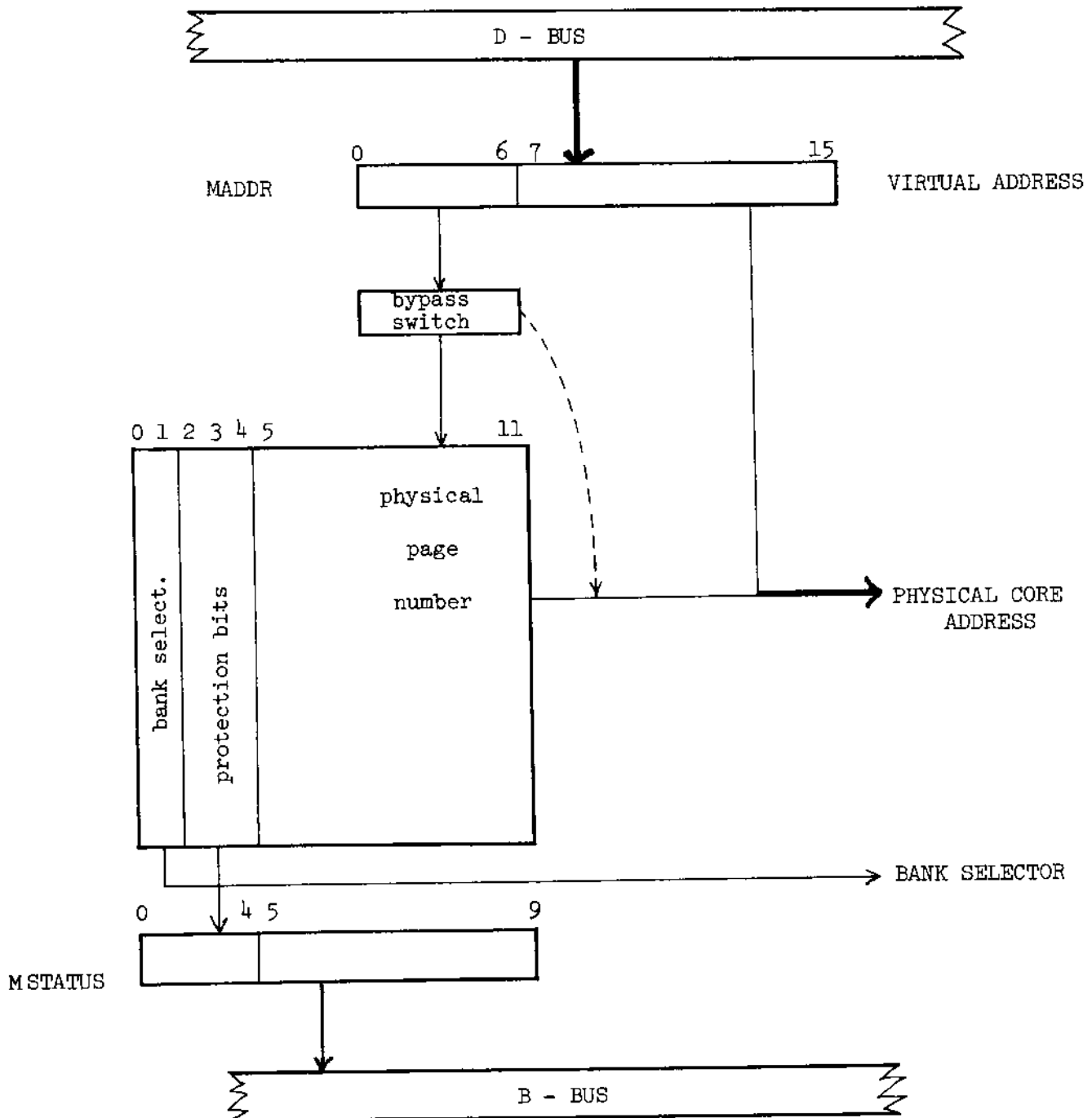


Figure J1 - The Hardware Map

The rightmost seven bits constitute the physical page number. While the map is inserted, every address reference to core sees its high order seven bits, the virtual page number, used as an index to the map where the seven-bit physical page number gets substituted, thus forming the final address in physical storage.

This substitution takes about 80-110 ns, but this delay is practically invisible: priority decoding is started at the same time as the map is referenced and proceeds in parallel. The interface with its priority logic settles in 80-90 ns so that the map only introduces a 0-20 ns additional delay.

Bits 2-4 are protection bits used to trap or protect certain pages.

- Bit 4 is write-protect: any attempt to write into a protected page is aborted and the write transformed into a read.
- Bit 3 is page fault, used by the firmware to detect an attempt to reference a page not owned by the process.
- Bit 2 is processor fault, used to prevent the APL machine from accessing a page being modified by the Operating System Processor.
- Bits 0-1 are used to select one out of four 65K banks. They are interpreted by a hardware bank selector.

All five special bits (0 through 4) get gated back to locations 0 through 4 of the memory status register everytime the map is referenced.

The operation mode (insert/remove) and contents of the map may be altered through the input-output registers of the processor, treating the map as an input-output device.

---

**K**

---

MAP CONTROL

## Appendix K - Map Control

The operation and contents of the map may be altered through the input-output registers of the processor, treating the map as an input-output device. The format is the following:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
IOADDR :	0	0	0	0	0	0	X	*	*	A	D	D	R	E	S	S
IODATA :	X	X	X	X	< . P A T T E R N . . >											

X = unused

**	Operation	Instructions
0 0	write in map.	IOADDR ← ADDRESS; IODATA ← PATTERN IO;
0 1	bypass map.	IOADDR ← 200B IO;
1 0	insert map.	IOADDR ← 400B IO;
1 1	read from map.	IOADDR ← ADDRESS OR 600B IO; REGISTER ← MAPDATA IO, PZ; * IO must be on * PZ is optional.

The map is referenced as IO device number 00 (bits 0-5 of IOADDR).  
A location within the map is addressed by bits 9-15 of IOADDR.

The timing is:

Read:

go signal	0
set read flipflop	+30
input ready	+200
Total cycle	500 ns (return of acknowledge)

Write:

go signal	0
set flipflop	+40
ready	+180

Insert or bypass: 100 ns

Timing restrictions:

For any command, an IO pulse must be given. In the case of a read, an IO must be given to acknowledge receipt of the data. Should



it not be given, the map will hang forever on the next access.

The use of PZ is optional, but recommended to insure a proper interlock between successive accesses.

Note that when any map command is executed, status bits are gated back to memory-status.

The commands:

In normal mode, only "write" is used to load the map with appropriate contents everytime a process is activated. This is described in the next section.

"Read" allows to examine the contents of any location within the map.

"Bypass" will cause the map to be bypassed. The map disappears from the address path and the processor accesses directly physical core.

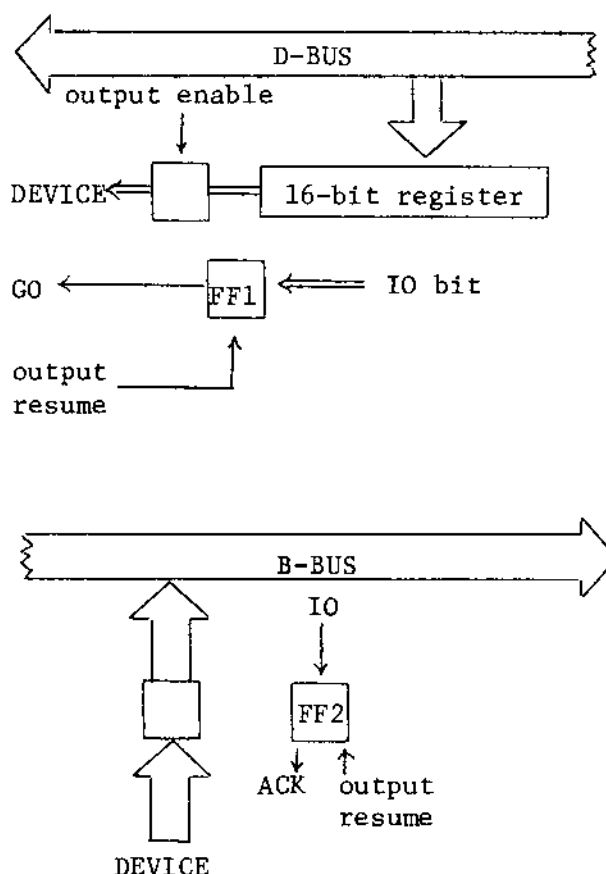
"Insert" inserts back the map in the address path.



INPUT - OUTPUT    REGISTERS

## Appendix L - Input-Output Registers

The 2 "registers" are each organized as follows:



The PZ bit 1) guarantees a 35 nsec delay.

- 2) halts the clock if FF1 is on and IO Register referenced on D-Bus, or FF2 is on and IO Register referenced on B-Bus.

PZ is decoded immediately.

IO sets the Flip-Flops only at the beginning of the next instruction, when data is finally entered from the D-bus into the destination register.

Note on receiving data from a device:

Data should be maintained until acknowledge (ACK) becomes active.

Output:

levels: data stays on.

## LISTING OF THE MICROPROGRAMMED APL INTERPRETER

This is a prototype only

\*INTERPRETER VERSION 7205.06

\*

DECLARE MACRO      DM ← DECLARE MACRO;

DM DR              ← DECLARE REGISTER;

DM DC              ← DECLARE CONSTANT;

\*THE REGISTERS :

DR ZERO            ← 0;

DR COUNTER        ← 1;

DR LINK            ← 2;

DR SPARED          ← 3;

DR MADDR           ← 4;

DR MDATA           ← 5;

DR ILOAD           ← 6;

DR IUDATA          ← 7;

DR C1              ← 8;

DR C2              ← 9;

DR C3              ← 10;

DR C4              ← 11;

DR MONITOR        ← 12;

DR PTR1            ← 13;

DR PTR2            ← 14;

DR PP              ← 15;

DR AL0             ← 24;

DR AL1             ← 25;

DR LOCALBASE      ← 26;

DR RETURN         ← 27;

DR JPR             ← 28;

DR STACK           ← 29;

DR FLAGS           ← 30;

DR VMA             ← 31;

DR D1              ← 16;\*              DBLE.

DR D2              ← 17;\*              DBLE.

DR D3              ← 18;\*              DBLE.

DR D4              ← 19;\*              DBLE.

DR AR0             ← 20;\*              DBLE.              RESULT & RIGHT OPRND,HI

DR AR1             ← 21;\*              DBLE.              RESULT & RIGHT OPRND,LO

DR VMD             ← 22;\*              BEXP IN FPP.

DR FREE            ← 23;\*              AEXP IN FPP.

\*

CONFIGURE GENERALAB SPARED,AR0,AR1,VMD,FREE,D1,D2,D3,D4;

CONFIGURE GENERALB C1,C2,C3,C4,RETURN,MONITOR,PP,AL0,AL1,PTR1,PTR2,  
LOCALBASE,JPR,FLAGS,VMA,STACK;

CONFIGURE MEMORY MADDR:4096,4,10;

\*

\*

## \*FLAGS:

\*-----

```

DM OPFNDF      ← FLAGS$0;
DM OPWIDF      ← FLAGS$1;
DM PARAMF      ← FLAGS$2;
DM DYADICF     ← FLAGS$3;
DM CHARF       ← FLAGS$4;
DM VECVECF     ← FLAGS$5;
DM SCASCAF     ← FLAGS$6;
DM RIPPLEF     ← FLAGS$7;
DM DIVYOF      ← FLAGS$12;
DM EXPOVERF    ← FLAGS$13;
DM EXPUNDERF   ← FLAGS$14;
DM INTOVERF    ← FLAGS$15;

```

## \*CONSTANTS:

```

DC OPFNDB      ← 1000000B;
DC OPWIDB      ← 0400000B;
DC PARAMB      ← 0200000B;
DC DYADICB     ← 0100000B;
DC CHARB       ← 0040000B;
DC VECVECB     ← 0020000B;
DC SCASCB      ← 0010000B;
DC RIPPLEB     ← 0004000B;
DC DIVBYOB     ← 000010B;
DC EXPOVERB    ← 000004B;
DC EXPUNDERB   ← 000002B;
DC INTOVERB    ← 000001B;

```

\*

```

DC OPFNDBM     ← 037777B;
DC OPWIDBM     ← 137777B;
DC PARAMBM     ← 157777B;
DC DYADICBM    ← 167777B;
DC CHARBM      ← 173777B;
DC VECVECBM    ← 175777B;
DC OPMASK      ← 170777B;
DC RIPPLEBM    ← 177377B;

```

\*

## \*MACROS:

\*-----

```

DM SKIP3       ← GOTO **8;
DM SKIP4       ← GOTO **10;
DM STATEIS1    ← FLAGS$1;
DM STATEIS2    ← FLAGS$0;
DM TURNOFF     ← FLAGS ← FLAGS AND ;
DM TURNON      ← FLAGS ← FLAGS OR ;
DM SETSTATE0   ← FLAGS ← FLAGS AND 037777B;
DM SETSTATE1   ← SETSTATE0 . FLAGS ← FLAGS OR OPWIDB;
DM SETSTATE2   ← SETSTATE0 . FLAGS ← FLAGS OR OPFNDB;
DM GLOBALBIT   ← VMD$4;
DM GO          ← LINK ← **4 . GOTO;
DM CALL        ← RETURN ← **4 . GOTO;
DM FBIT        ← VMD$1;
DM READIT      ← LINK ← **4 . GOTO READ;

```

```

DM WRITEIT      ← LINK ← **4 . GOTO WRITE;
DM EXITIN1      ← GOTO OUTIN1;
DM EXITIN2      ← GOTO OUTIN2;
DM MIXEDOPBIT   ← OPR$6;
DM EXITTORETURN ← GOTO OUT;
DM SAVEAR       ← C3 ← AR0 . VMA ← AR1 ;
DM SAVEAL       ← C1 ← AL0 . C2 ← AL1;
DM RESTOREAR    ← AR0 ← C3 . AR1 ← VMA;
DM RESTOREAL    ← AL0 ← C1 . AL1 ← C2;
DM CROSSSTORE  ← AR0 ← C1 . AR1 ← C2;
*
*
```

\*

\*MASKS:

\*-----

DC MASK2	← 140000B;
DC MASK3	← 160000B;
DC MASK4	← 170000B;
DC MASK5	← 174000B;
DC MASK8	← 177400B;
DC ARGMASK	← 060000B;
DC KILLBIT0	← 077777B;
DC KILL3	← 017777B;
DC KILL4	← 007777B;
DC KILL5	← 003777B;
DC KILL8	← 000377B;
DC KILL10	← 000077B;
DC KILLBIT4	← 173777B;
DC SIGNBITM	← 177577B;
DC OPDONEM	← 161777B;

\*

\*CONSTANTS:

\*-----

DC FUZZM	← 177776B;
DC FLOATINGONE	← 500B;
DC HEADERSIZE	← 4;
DC HISTORYSIZE	← 7;
DC FLOATINGB	← 040000B;
DC GARBAGEB	← 010000B;
DC FLOATINGM	← 137777B;
DC BIT0	← 100000B;
DC BIT1	← 040000B;
DC BIT2	← 020000B;
DC BIT3	← 010000B;
DC BIT4	← 004000B;
DC BIT5	← 002000B;
DC BIT9	← 000100B;
DC BLOP	← 177775B; *
DC FONTABLE	← 177777B;
DC GLOBALBASE	← 177771B;
DC SIGNBIT	← 000200B;

\*

BLOCK ALLOCATION PIR IN GFRZ



## \*DESCRIPTORS

\*-----

## \*EXECOPS:

DC LPARENCODE   ← 122002B;\*       X2  
 DC LBRACKETC   ← 122003B;\*       X3  
 DC LBRACECODE   ← 122001B;\*       X1  
 DC LARROWCODE   ← 122005B;\*       X5  
 DC SEMICLNCODE  ← 122006B;\*       X6

## \*RIGHT DELIMS:

DC RPARENCODE   ← 124002B;\*       R2  
 DC RBRACKETC   ← 124000B;\*       R0

## \*TOKENS:

DC RBRACECODE   ← 160000B;  
 DC APCODE       ← 100000B;  
 DC LDCODE       ← 122004B;\*       X4  
 DC NULLCODE     ← 100000B;  
 DC LNCODE       ← 140000B;  
 DC FCCODE       ← 160000B;  
 DC UVCODE       ← 170000B;  
 DC PMCODE       ← 120000B;  
 DC RDELIMCODE   ← 124000B;  
 DC PHCODE       ← 110000B;  
 DC PPCODE       ← 140000B;  
 DC VECTORCODE   ← 100001B;

\*\*\*\*\*

## \*OPERATORS:

DC BACKSLASH     ← 121127B;  
 DC SLASHCODE     ← 121126B;  
 DC EQUALCODE     ← 120220B;  
 DC NEQUALCODE    ← 120221B;  
 DC TRANSPCODE    ← 121130B;  
 DC DOTCODE       ← 121031B;

\*

```

*****
*   U T I L I T Y           R O U T I N E S   *
*****
*****
*PUSH*
*****
PUSH:      STACK ← STACK - 1;
           VMA ← STACK;
           GOTO WRITE;

*****
*READ*
*****
READ:      MADDR ← VMA AND 007777B MR;
           VMD ← MDATA J;*65K IS MAPPED INTO 4K FOR SIMULATION
WRITE:     MADDR ← VMA AND 007777B;
           MDATA ← VMD MW , J;

*****
*POP*
*****
POP:       VMA ← STACK;
           STACK ← STACK + 1;
           GOTO READ;

*****
*LOGICALVERIFY*
*****
LOGICALVERIFY: GOTO ERROR IF VMD # 0;*NON LOGICAL WHERE LOGICAL XPECTD
               GOTO 0 IF SPARED$R = 0 IX;*      IS IT 0? ... RETURN
               VMD ← SPARED EOR FLOATINGONE;*    IS IT A 1 ?
               GOTO ERROR IF VMD # 0;*NONLOGICAL FOUND
               ZERO ← ZERO J;

*THIS ROUTINE EXPECTS A FLOATING NUMBER IN SPARED AND VMD.
*NO REGISTER GETS CHANGED, UNLESS ERROR EXIT OCCURS
*****
*****
*EXITS*
*****
OUTIN1:    SETSTATE1;
           GOTO PARSENEXT;
OUTIN2:    SETSTATE2;
           GOTO PARSENEXT;
NOTYET    : GOTO *;
ERROR:     GOTO *;
OUT:      LINK ← RETURN;
OUT1:     ZERO J;

*****
*LOADAR*
*****
LOADAR:    READIT;
           AR0 ← VMD;
           VMA ← VMA + 1;
           READIT;
           LINK ← RETURN;
           AR1 ← VMD J;

```

```

LOADAL:      READIT;
              AL0 ← VMD;
              VMA ← VMA + 1;
              READIT;
              LINK ← RETURN;
              AL1 ← VMD J;

*****  SWAPREGISTERS      LINK-ROUTINE  *****
SWAPREGISTERS: SPARED ← AR0;
               AR0 ← AL0;
               AL0 ← SPARED;
               SPARED ← AR1;
               AR1 ← AL1;
               AL1 ← SPARED J;

*
*-----
* AR1 POINTS TO THE BLOCK.
* MAKE A COPY OF IT
* C2 RETURNS POINTER TO NEW BLOCK
*-----
COPYBLOCK:    VMA ← AR1 + 1; *           GET SIZE
               GO READ;
               C1 ← VMD; *           TOTAL # WORDS= COUNTER
               D1 ← RETURN;
               CALL GETBLOCK; *       C2 RETURNS PTR TO WORD 0
               RETURN ← D1;
               PTR1 ← AR1;
               PTR2 ← C2;
               VMA ← AR1;
               GO READ;
               VMD ← VMD AND KILLBIT0;
               VMA ← C2;
               GO WRITE;
               GOTO XFER1;
XFERLOOP:     VMA ← PTR1;
               GO READ;
               VMA ← PTR2;
               GO WRITE;
XFER1:        C1 ← C1 - 1;
               GOTO OUT IF C1 = 0;
               PTR1 ← PTR1 + 1;
               PTR2 ← PTR2 + 1;
               GOTO XFERLOOP;

*
*-----
* VMA POINTS TO WORD 0 OF BLOCK
*
RELEASEBLOCK: READIT;
               VMD ← VMD OR GARBAGE;
               LINK ← RETURN;
               GO TO WRITE;
*ANY ALGORITHM CAN BE SUBSTITUTED LATER.
*

```

```

*
*
*LINE NUMBER IS IN FREE.
FIRSTLINE:    FREE ← 1;
              GOTO STARTLINE;

*
NEXTLINE:     GOTO DISMISS IF FREE = 0;*   PROGRAM MODE?
              FREE ← FREE + 1;*           INCREMENT LINE #
              VMA ← STACK + 3;
              READ1;*                       GET MAX LINE #
              C1 ← FREE EOR -1;
              VMD ← C1 + VMD P1;*         (MAX#) - (LINE#)
              GOTO RETURN IF VMD$0 = 1;*   <0?... RETURN!
STARTLINE:    VMD ← FREE OR LNCODE;*       NEW LN FOR STACK
              GO PUSH;
              VMA ← STACK + 5;*           GET NEXT HS ENTRY
              READ1;*                       GET LINE TABLE PIR
              VMA ← VMD + FREE;
              READ1;*                       ENTRY FOR LINE # FREE
              PP ← VMD;*                   RESET PP TO END OF LINE
CLEAR:        FLAGS ← 0;
PARSENEXT:    LINK ← DECYPHER;
GETTOKEN:     VMA ← PP;
              PP ← PP + 1;
              GOTO READ;

*
*
*
CALCMODE:     VMD ← FREE EOR LNCODE;
              LINK ← CLEAR;
              GOTO PUSH;

*
*
DISMISS:      GOTO CALCMODE;

*
*
DECYPHER:     GOTO SCALAR IF VMD$0 = 0;*   0--- ?
              GOTO D10 IF VMD$1 = 0;*     10-- ?
              GOTO D110 IF VMD$2 = 0;*     110- ?
              GOTO RBRACE IF VMD$3 = 0;*   1110 ?
              GOTO ERROR;*                 1111 UNUSED
D10:          GOTO D100 IF VMD$2 = 0;*     100- ?
              GOTO OPERATOR IF VMD$3 = 0;* 1010 ?
              GOTO PARSENEXT;*             1011 NOJP
D110:         GOTO 10D IF VMD$3 = 0;*      1100 ?
              GOTO COMMAND;*              1101
D100:         GOTO ARRAY IF VMD$3 = 0;*    1000 ?
              GOTO PHANTOM;*              1001
*VMD CONTAINS THE TOKEN
*A JUMP TABLE IS NOT APPROPRIATE BECAUSE OF THE 5 OPERATIONS
*NEEDED TO SHIFT A DESCRIPTOR INTO THE RIGHT PART OF THE LINK
*

```

```

SCALAR:      AL0 ← VMD;*          LOAD INTO AL
             GO GETTOKEN;
             AL1 ← VMD;*          2ND WORD OF SCALAR
OTYPECHECK:  GOTO ERROR IF STATEIS1 = 1;* MISSING OPERATOR ?
             GOTO BINOPER IF STATEIS2 = 1;*2ND OPERAND FOUND ?
             AR0 ← AL0;*          STATE IS 0 : RIGHT OPND
             AR1 ← AL1;
             EXITIN1;*           OPERAND FOUND

```

\*

```

ARRAY:      AL0 ← VMD;
             PP ← PP - 1;
             AL1 ← PP;*AP
             VMA ← PP + 1;
             GO READ;
             PP ← PP + VMD;
             GOTO OTYPECHECK;
             GOTO OTYPECHECK;

```

\*

\*\*\*\*\* GET PHANTOM ROUTINE \*\*\*\*\*

\*\*\*\*

\* VMD CONTAINS THE PHANTOM DESCRIPTOR.

\*THE PHANTOM IS OBTAINED FROM THE APPROPRIATE SEGMENT

\*AND LOADED INTO AL

\*

```

GETPHANTOM:  SPARED← VMD AND KILL5;*      GET PHANTOM NUMBER
             SPARED L1;*                  DOUBLE IT
             SPARED ← SPARED EOR -1;*     COMPLEMENT FOR SUBSTRACT
             VMA ← GLOBALBASE;
             SKIP1 IF GLOBALBIT = 1;*     LOCAL ?
             VMA ← LOCALBASE;
             VMA ← VMA + SPARED P1;*     POINT TO THE PMS ENTRY
ACCESSPHANTOM: GO READ;
             AL0 ← VMD;*                  LOAD THE ENTRY INTO AL
             VMA ← VMA + 1;
             READIT;
             LINK ← RETURN;
             AL1 ← VMD J;

```

\*REGISTERS MODIFIED: VMA, SPARED, VMD.

\*\*\*\*\*

PHANTOM: CALL GETPHANTOM;\* LOAD AL FROM PH IN VMD

```

PHIPROCESSOR: GOTO OTYPECHECK IF AL0$0 = 0;* 0-
              GOTO P10 IF AL0$1 = 0;* 10-- ?
              GOTO P110 IF AL0$2 = 0;* 110- ?
              GOTO FCNCALL IF AL0$3 = 0;* 1110 ?
              GOTO ERROR;* 1111 UV 1
P10:          GOTO P100 IF AL0$2 = 0;* 100- ?
              GOTO PM IF AL0$3 = 0;* 1010 ?
              GOTO DEVICE;* 1011
P110:         GOTO PPR IF AL0$3 = 0;* 1100 ?
              GOTO ERROR;* 1101 UNUSED
P100:         GOTO OTYPECHECK IF AL0$3 = 0;* 1000 ?
              GOTO ERROR;* 1001 UNUSED

```

\*

```

FCNCALL:      C1      + AL1 AND ARGMASK;*      EXTRACI # ARGS
              GOTO F0  IF OPNDF = 0;*          STATE 0- ?
              GOTO F10 IF OPWDF = 0;*          STATE 10 ?
              GOTO ERROR;*                     STATE 11 IS ILLEGAL
F0:           GOTO PARAMCHECK IF OPWDF = 0;*    STATE 00 ?
              GOTO ERROR IF C1 = 0;*          STATE 01 .SUPERFLUOUS ARG.
              GOTO PARAMCHECK;*
F10:          GOTO MONOPER IF C1 # 0;*          STATE IS 2 AND NO ARGS
              GOTO ERROR IF AL1$0=0;*        ARG OR RESULT MISSING
*
PARAMCHECK:   GOTO NOPARAM IF PARAMF=0;
              TURNOFF PARAMM;*              THERE WERE PARAMS
              GOTO PARAMMATCH;
NOPARAM:      GOTO NODEFER IF C1 = 0;*          ONLY ONE ARGUMENT
              CALL DEFER;
              STACK + STACK + 1;*OPR HAS BEEN ENTERED:DELETEIT
NODEFER:      FREE + 0;*                      ACTUAL PARAM #
PARAMMATCH:   C2      + AL1 AND KILL3;*        FORMAL PARAM #
              C3      + C2 EOR FREE;*          MATCH ?
              GOTO ERROR IF C3#0;*            WRONG # OF PARAMS
              GOTO NUFRAME IF AL1$1 = 0;*      PROCEED FOR 0-1 ARG
              OPR      + AL0;*                2 ARG: SAVE FC IN OPR
              EXITIN2;
*
*NUFRAME
*-----
*AL CONTAINS THE DOUBLE TOKEN FOR FUNCTION CALL
*C1 CONTAINS THE NUMBER OF ARGUMENTS IN BITS 1-2
*C2 CONTAINS THE NUMBER OF PARAMETERS, RIGHT JUSTIFIED.
NUFRAME:      VMA      + FCNTABLE      ;*      GET FUNCTION TABLE
              GO READ;
              AL0      + AL0 AND KILL4;*      EXTRACT FCN #
              C3 + AL0 L1;
              C3 L1;*                          *4=INDEX TO TABLE
              C4      + C3 + VMD;*          SAVE PTR TO ENTRY
              VMA + C4;*ACCESS 1ST ENTRY (FOR BINFC'S BENEFIT!)
              READIT;
              AL1 + VMD;*AGAIN, THIS IS FOR A BINFC.
              VMA + C4+1;*                  ACCESS 2ND ENTRY
              READIT;
              C1 + AL1 AND ARGMASK;
              C2 + AL1 AND KILL3;
*VMD CONTAINS THE TOTAL PHANTOM NUMBER.
*COMPUTE THE ADJUSTED PMS SIZE = TOTAL# - ARG# - PARAM#
*THIS IS THE NUMBER OF REMAINING ENTRIES THAT HAVE TO BE
*CREATED IN THE PMS.
*NOW C1 CONTAINS #ARG IN BITS 1-2
*      C2 CONTAINS #PARAMS.
              D1 + STACK - 2;*RESULT ADDRESS
              SKIP3 IF C1 = 0;*              NO ARGUMENTS ?
              C2      + C2 + 1;
              SKIP1 IF C1$1=0;*              ONLY ONE ARG
              C2      + C2 + 1;*              C2 CONTAINS ARG#

```

	SPARED ← C2 EOR 177777B;	
	C3 ← VMD + SPARED P1;*	ADJUSTED PMS SIZE
	C3 L1;*	DOUBLE IT=#WORDS
	C1 ← VMD L1;*	SAVE TOTAL PMS WORD SIZE
	VMD ← UVCODE;	
	VMA ← STACK;	
	SPARED ← C3 EOR 177777B;	
	STACK ← STACK + SPARED P1;*	SET NEW STACK PTR
	C2 ← STACK;*	SAVE FOR FUTURE RESULT PTR
	LINK ← ENTERUV;*	PREPARE FOR LOOP
ENTERUV:	GOTO NUHISTORY IF C3 = 0;*	C3 IS COUNTER-END TEST
	VMA ← VMA - 2;	
	C3 ← C3 - 2;*	DECREMENT COUNTER
	GOTO WRITE;*	NO MACRO!(LINK IS SET)
*		
NUHISTORY:	VMD ← AL0;*	STACK THE DOUBLE FC TOKEN
	GO PUSH;	
	VMD ← AL1;	
	GO PUSH;	
	VMA ← C4 + 2 ;*	3RD TABLE ENTRY(4N+2)
	READIT;	
	C3 ← VMD;*	SAVE LINE TABLE PTR
	GO PUSH;	
	VMA ← C3 - 1;*	NBR OF LINES OF LTBLE
	READIT;	
	GO PUSH;	
	VMD ← D1 ;*	EVENTUAL RESULT POINTER
	GO PUSH;	
	VMD ← LOCALBASE;*	STORE LOCALBASE
	GO PUSH;	
	SPARED ← STACK + 4;*	2ST WORD OF HIST SEGMENT
	LOCALBASE ← C1 + SPARED;*	C1 HAS PMS 'ORD SIZE
*LOCALBASE NOW	POINTS TO THE SECOND WORD OF THE NEW PMS	
	VMD ← PP;	
	GO PUSH;*	SAVE PP
*		

```

ENTERLABELS:  VMA = C4 + 3;*          4TH FT ENTRY=4N+3
               READIT;
               GOTO FIRSTLINE IF VMD = 0;* NO LABELS ?
               VMA = VMD;*           ACCESS LABEL TABLE
               READIT;
               C3 = VMD;*           #LABELS = COUNTER
               PTR1 = VMA - 2;*      PREPARE FOR LOOP
LABLEENTER:    GOTO FIRSTLINE IF C3 = 0;* TEST COUNTER FOR ENDOFTTEST
               PTR1 = PTR1 + 3;*     3 WORD ENTRIES.
               VMA = PTR1;*         PHANTOM #
               READIT;
               SPARED = VMD L1;*     SAVE ITS DOUBLE
               VMA = VMA + 1;*       NEXT 2 WORDS ARE VALUE
               READIT;
               AL0 = VMD;
               VMA = VMA + 1;
               READIT;
               AL1 = VMD;
               SPARED = SPARED EOR 177777B;
               VMA = LOCALBASE + SPARED P1;*ACCESS PH IN THE PMS
               VMD = AL0;*          ENTER ITS VALUE
               GO WRITE;
               VMA = VMA + 1;
               VMD = AL1;
               C3 = C3 - 1;*        DECREMENT COUNTER
               LINK = LABLEENTER;
               GOTO WRITE;

```

★



```

OPERATOR:      SKIP1 IF STATEIS1 = 1;*      STATE 0 IS ERROR
               GOTO RDELIM IF VMD$4 = 1;
               C1 ← FLAGS AND MASK2;
               GOTO ERROR IF C1 = 0;*      MISSING OPERAND
               GOTO OPSTATE1 IF STATEIS2=0;
               C4 ← VMD;*                  STATE IS 2.SAVE THE OPERATOR
               GOTO MONOPER;

*
OPSTATE1:      GOTO EXECOP IF VMD$5=1;
               OPR ← VMD;*                  NORMAL OPERATOR
               EXITIN2;

*
*A RIGHT DELIMITER STARTS A NEW EXPRESSION LEVEL-THEY ARE:  )]
RDELIM:        RETURN ← PUSHDEL;
               C2 ← VMD;
               GOTO ERROR IF STATEIS1 = 1;
               GOTO DEFER IF STATEIS2 = 1;

*
PUSHDEL:       VMD ← C2;
               LINK ← CLEAR;
               GOTO PUSH;

*
*EXECOPS ARE :   RARROW ( [ LD LBRACE ← ;
EXECOP:         LINK ← VMD AND KILL10;*      LAST 6 BITS = OPCODE
               LINK L1;*                    DOUBLE IT
               EXECUTE XOPTABLE IX;

*
ORIGIN 1200B;
XOPTABLE:      GOTO BRANCH;*                0
               GOTO LBRACE;*                1
               GOTO LPAREN;*               2
               GOTO LBRACKET;*             3
               GOTO LD;*                   4
               GOTO ASSIGN;*               5
               GOTO SEMICLN;*              6

```

```

*****
DEFER:      D E F E R      R O U T I N E      *****
            C1  ← AR0 AND MASK4;*      EXTRACT DESCRIPTOR
            C1  ← C1 EOR APCODE;*      CHECK FOR ARRAY
            GOTO STACKOPRAR IF C1 #0;*      NOT AN ARRAY ?
*THE ABOVE(NOT ARRAY) IS AN ERROR IF PARAMF#1 .
            GOTO STACKOPRAR IF AR1=0;*      A NULL ?
            VMA ← AR1 + 3;*      DEFERMENT FIELD
            READI;
            GOTO STACKOPRAR IF VMD #0;*      ALREADY DEFERRED ?
            VMD ← STACK - 2;
            GO WRITE;*      ENTER SAVEDSTACK
***** ROUTINE & *****
STACKOPRAR: VMD ← AR1;
            GO PUSH;
STACKAR:    VMD ← AR0;
            GO PUSH;
            VMD ← OPR;*      ATTENTION : OPR GETS PUSHED
            LINK ← RETURN;
            GOTO PUSH;
*****

```

```

BRANCH:      GO GETTOKEN ;*          NEXT TOKEN MUST BE LD
             VMD ← VMD EOR LDCODE;
             GOTO ERROR IF VMD #0 ;*      ILLEGAL BRANCH SYNTAX
*NOTHING SHOULD BE LEFT ON THE STACK EXCEPT LN
*CHECK IT

             GO POP;
             C1 ← VMD AND MASK4;
             C1 ← C1 EOR LNCODE;
             GOTO ERROR IF C1 # 0 ;*THERE SHOULD BE AN LN ON STACK
             FREE ← VMD AND KILL4 ;*      GET LINE #
             GOTO ERROR IF FREE = 0 ;*      NO BRANCH IN CALC MODE
             GOTO TESTSIGN IF AR0$0=0 ;*    SCALAR?
             C1 ← AR0 EOR NULLCODE;
             GOTO NEXTLINE IF C1 = 0 ;*      NULL ?
             C1 ← AR0 AND MASK4 ;*          EXTRACT DESCRIPTOR
             C1 ← C1 EOR APCODE;
             GOTO ERROR IF C1#0 ;*          ILLEGAL OPERAND FORBRCH
             GOTO ERROR IF AR0$4=1 ;*      ARRAY MUST BE NUMERIC
             VMA ← AR1 + 1 ;*              ARRAY SIZE
             READIT;
             VMA ← AR1 + VMD;
             VMA ← VMA - 2;
             CALL LOADAR;
TESTSIGN:    GOTO ERROR IF AR0$8=1 ;*      NO BRANCH TO NEGATIVE
             CALL FIX ;*                  FIXED INTEGER IN AR1
             GOTO FRETURN IF AR0#0 ;*      > 65K ?
             GOTO FRETURN IF AR1=0;
             GOTO FRETURN IF FLAG$15=1 ;*  OVERFLOW OF INTEGER
             SPARED ← AR1 EOR 177777B;
             VMA ← STACK + 3 ;*          MAX # LINES
             READIT;
             SPARED ← VMD + SPARED P1 ;*  MAX# - LINE#
             GOTO FRETURN IF SPARED$0=1 ;*  OUT OF BOUNDS ?
             FREE ← AR1 ;*                RESET LINE #
             GOTO STARTLINE;

*
*
*
*
SEMICLN:    OPR ← VMD;
             CALL DEFER;
             STACK ← STACK -2;
             LINK ← CLEAR;
             VMD ← OPR;
             GOTO PUSH;

*

```

```

*STACK IS EMPTY, LINE # IS IN FREE
*STACK POINTER POINTS TO WORD 7 OF HISTORY SEGMENT(FORMER PP)
FRETURN:      VMA  ← STACK  + HISTORYSIZE;*      TOP OF PMS
ERASELOOP:    SPARED ← VMA EOR -1;
              C1 ← LOCALBASE + SPARED P1;*  LOCALBASE - ADDR
              GOTO LFRZCLEAN IF C1$0=1;*      < 0 ?
              GO READ;*                      EXAMINE PMS ENTRY
              C1 ← VMD AND MASK4;*           GET DESCRIPTOR
              C1 ← C1 EOR APCODE;*           AN ARRAY ?
              GOTO NOVECTOR IF C1 #0;
              C1 ← VMD EOR NULLCODE;*        NULL ?
              CALL RELEASEBLOCK IF C1#0;
NOVECTOR:     VMA ← VMA + 2;*                NEXT PMS ENTRY
              GOTO ERASELOOP;

*
LFRZCLEAN:    GO POP;
              PF ← VMD;*                    RESTORE PP
              C2 ← LOCALBASE + 2;*          NEXT STACK POINTER
              GO POP;
              LOCALBASE ← VMD;*             RESTORE LOCALBASE
              GO POP;
              C1 ← VMD;*                   RESULT POINTER
              STACK ← STACK +2;*           SKIP 2 HISTORY ENTRIES
              GO POP;
              C3 ← VMD;*                   RESULT-ARG-PARAM
              GOTO NORESULT IF VMD$0=0;*    NO RESULT ?
              VMA ← C1;*                   RESULT ACCESS IT
              CALL LOADAR;
              STACK ← VMA + 1;

*NOTE IF AN ARRAY IS IN AR, IT HAS BETTER BE MARKED FRESH.
              C3 ← C3 AND ARGMASK;*        GET # ARGUMENTS
              GOTO UNSTACK IF C3 = 0;*      SAME AS C IF NO ARGS
              STACK ← C2;*                 RESTORE STACK PTR
              SETSTATE1;
              GOTO PARSENEXT;

*
NORESULT:     STACK ← C2;
              GO GETTOKEN;*               NEXT TOKEN MUST BE LD
              VMD ← VMD EOR LDCODE;
              GOTO ERROR IF VMD#0;*        NO RESULT FCN
              RETURN ← NEXTLINE;
              GOTO ENDLIN;*               RESET LINE #

*

```

```

LBRACE:      GOTO ERROR IF PARAMF = 0;*   MISSING RIGHT BRACE
              GOTO PARSENEXT;

*
LPAREN:      GO POP;
              VMD ← VMD EOR RPARENCODE;
              GOTO ERROR IF VMD ≠ 0;*
UNSTACK:     VMA ← STACK;*               ILLEGAL (
              READI;                     COPY 10POFSTACK FOR OPER
              C1 ← VMD AND MASK5;
              C1 ← C1 EOR RDELIMCODE;
              EXITINI IF C1 = 0;*        RIGHT DELIM(INCLUDES LN)
              C1 ← VMD AND MASK4;
              C1 ← C1 EOR LNCODE;*       TEST FOR ENDOFLINE
              EXITINI IF C1 = 0;

*IT IS A NORMAL OPERATOR OR A FUNCTION CALL
POPEM:       AL0 ← AR0;
              AL1 ← AR1;*               TRANSFER AR TO AL
              GO POP;
              OPR ← VMD;
              VMA ← STACK;
              CALL LOADAR;
              STACK ← STACK + 2;
              GOTO BINOPER;

*

```

```

LBRACKET:      OPR ← VMD;*                LOAD OPR
                VMA ← STACK;*             EXAMINE TOPOFSTACK
                READIT;
                C1 ← VMD EOR SEMICLNCODE;
                EXITIN2 IF C1 = 0;*        SYNTAX OK
                C1 ← VMD EOR RBRACKETCODE;
                GOTO ERROR IF C1 ≠ 0;
                EXITIN2;

*
*
*
LD:            RETURN ← PRINTAR;
***** E N D   L I N E       R O U T I N E *****
ENDLINE:      GO POP;
                FREE ← VMD AND KILL4;*     EXTRACT LINE #
                C1 ← VMD AND MASK4;
                C1 ← C1 EOR LNCODE;*       SPECIAL CODE NEEDED FOR LN
                LINK ← RETURN;
                GOTO 0 IF C1 = 0 IX;*      EXIT IF LN FOUND
*OTHERWISE, FALL THROUGH :
*****
                C1 ← VMD EOR SEMICLNCODE;
                GOTO ERROR IF C1 ≠ 0;
                CALL PRINTAR;
                VMA ← STACK;
                CALL LOADAR;
                STACK ← STACK + 2;*        UPDATE POINTER
                GOTO LD;*                  ITERATE SINCE ; FOUND

*
DEVICE:      GOTO NOTYET;
*

```

\*-----\*

\* T ~ A \*

\*\*\*\*\*

\*

\*AT THIS POINT, A MUST HAVE BEEN EVALUATED.

\*

\*C3 CONTAINS PMS ENTRY !!

```

ASSIGN:      GO GETTOKEN;*          NEXT TOKEN MUST BE PH OR I
              C1 ~ VMD AND MASK4;
              C1 ~ C1 EOR PHCODE;
              GOTO INDEXCHECK IF C1 # 0;* NOT A PH

```

\*IT IS A PH: LOOK IT UP

```

LOOKITUP:    CALL GETPHANTOM;*      PH CONTENTS RETURNED IN AL
              C3 ~ VMA - 1;*        PHANTOM POINTER
              VMD ~ AL0 AND MASK4;*  EXTRACT 4 BIT DESCRIPTOR
              C1 ~ VMD EOR FCCODE;*  IS IT AN FC ?
              GOTO ERROR IF C1 = 0;* NO ~ TO FCN NAME
              SETSTATE1;*           ANTICIPATE MIDDLE OF LINE ~
              C1 ~ VMD EOR PMCODE;*  IS IT A PARAMETER ?
              SKIP2 IF C1 # 0;
              VMA ~ AL1;*           IT WAS A PM ~ GET ADDRESS FIELD
              GOTO LOOKITUP;
              C1 ~ VMD EOR PPCODE;*  PP ?
              GOTO PPASSIGN IF C1 = 0;

```

\*IT IS AP,UV,S,IOD,NULL.

```

              C1 ~ VMD EOR APCODE;
              GOTO RIGHTRELEASE IF C1 # 0;* AP ?

```

\*IT IS AN AP :

```

              C1 ~ AL0 EOR NULLCODE;
              GOTO RIGHTRELEASE IF C1 = 0;* NULL?

```

\*NON NULL AP WAS IN T.

```

OLDVALUERELEASE: VMA ~ AL1 + 3;*    ACCESS SAVEDSTACKFIELD
                  GO READ;
                  GOTO NOTDEFERRED IF VMD = 0;* DEFERRED?
                  VMA ~ AL1;*       IT WAS DEFERRED
                  GO READ;
                  VMD ~ VMD OR FLOATINGB;* TURN F BIT ON
                  GOTO RIGHTRELEASE;

```

\*

```

NOTDEFERRED:    C1 ~ AR0 AND MASK4;
                  C1 ~ AR0 EOR APCODE;
                  GOTO TNEQA IF C1 # 0;
                  C1 ~ AL1 EOR AR1;*  A IS ARRAY
                  GOTO TNEQA IF C1 # 0;
                  C1 ~ AL0 EOR AR0;
                  GOTO MIDDLEOFLINE IF C1 = 0;* IDENTICAL T AND A PIRS
TNEQA:          VMA ~ AL1;
                  CALL RELEASEBLOCK;

```

\*

```

RIGHTRELEASE:   VMD ~ AR0 AND MASK4;*  EXTRACT 4 BIT DESCRIPTOR
                  C1 ~ VMD EOR APCODE;
                  GOTO ENTERIT IF C1 # 0;

```

\*IT IS AN ARRAY:

```

        C1 ← AR0 EOR NULLCODE;
        GOTO ENTERIT IF C1 = 0;*  NULL ?
*REAL ARRAY:
        VMA ← AR1;
        C1 ← COPYIT;
        C2 ← **4;
        GOTO RELEASETEST;
*RELEASABLE. DON T CHANGE AR. MARK IT FRESH.
        VMA ← AR1;
        GO READ;
        VMD ← VMD AND FLOATINGM;*  KILL FBIT
        GO WRITE;
        GOTO ENTERIT;
*NOT RELEASABLE : CREATE A COPY OF ARRAY . AR1 POINTS TO THE ARRAY.
COPYIT:
        CALL COPYBLOCK;
        AR1 ← C2;
        VMA ← AR1 + 3;
        GO READ;
        VMD ← 0;
        GO WRITE;*          ZERO SAVEDSTACK
        VMA ← AR1;
        GO READ;
        VMD ← VMD AND FLOATINGM;*  ZERO FBIT
        GO WRITE;
ENTERIT:
        VMA ← C3;
        VMD ← AR0;
        GO WRITE;
        VMA ← VMA + 1;
        VMD ← AR1;
        GO WRITE;
MIDDLEOFFLINE:
        GO GETTOKEN;
        C1 ← VMD EOR LDCODE;
        RETURN ← NEXTLINE;
        GOTO ENDLINE IF C1 = 0;
        GOTO DECYPHER;*          MIDDLE OF LINE ←
*
*T WAS NOT A PHANTOM !....
INDEXCHECK:
        C1 ← VMD EOR RBRACKETCODE;*  IS IT A ] ?
        GOTO ERROR IF C1 ≠ 0;*  ILLEGAL ←
*IT IS AN INDEXED ASSIGNMENT:
        CALL DEFER;
        VMD ← RBRACKETCODE;
        GO PUSH;
        SETSTATE0;
        GOTO PARSENEXT;
*
PPASSIGN:
        GOTO NOTYET;
*
```



```

NULL:      AR0 ← NULLCODE;
           TURNON SCASCAB;
           AR1 ← ZERO;
           GOTO OPRETURN;

*
*
*
RBRACE:    VMD ← VMD AND KILL4;*      EXTRACT DISPLACEMENT
           VMA ← PP + VMD;* ACCESS PHANTOM
           GO READ;
           C1 ← VMD AND MASK4;*VERIFY PH
           C1 ← C1 EOR PHCODE;
           GOTO ERROR IF C1 ≠ 0;
           CALL GETPHANTOM;
           VMD ← AL0 AND MASK4;
           VMD ← VMD EOR FCCODE;
           GOTO ERROR IF VMD ≠ 0;
           VMD ← AL1 AND 060000B;* EXTRACT # ARGS
           GOTO NOARGCASE IF VMD = 0;

*1 OR 2 ARGS
           GOTO MONOPER IF STATEIS2 = 1;
           GOTO ERROR IF STATEIS1 = 0;*STATE IS 0
           CALL DEFER;
           STACK ← STACK + 1;*      POP THE OPR PUSHED
           GOTO PARAMINIT;
NOARGCASE: GOTO ERROR IF STATEIS1 = 1;
           CALL DEFER IF STATEIS2 = 1;
PARAMINIT: FREE ← 0;*              PARAMETER COUNTER
           TURNON PARAMB;
*VERIFY NOW THAT THE NEXT TOKEN IS A PHANTOM AND NOT AN FC
VERIFYPHNOTFC: GO GETTOKEN;
           C1 ← VMD AND MASK4;* EXTRACT DESCRIPTOR
           C1 ← C1 EOR PHCODE;
           GOTO ERROR IF C1 ≠ 0;* ILLLEGAL PARAMETER
           CALL GETPHANTOM;
           C2 ← VMA - 1;
           C1 ← AL0 AND MASK4;
           C1 ← C1 EOR FCCODE;
           GOTO ERROR IF C1 = 0;

*
*
*
*
**IN PARAM MODE, FREE CONTAINS THE # OF PARAMETERS
NEXTPARAMETER: FREE ← FREE + 1;*      INCREMENT COUNT
           VMD ← AL0 EOR PMCODE;* WAS IT A PM ?
           GOTO FIRSTTIME IF VMD ≠ 0;
           VMD ← AL1;*              GET POINTER
           SKIP1;
FIRSTTIME: VMD ← C2;
           GO PUSH;
           VMD ← PMCODE;
           GO PUSH;

```

```

GO GETTOKEN;
C1 ← VMD EOR SEMICLNCODE;
GOTO VERIFYPHNOTFC IF C1 = 0;
C1 ← VMD EOR LBRACECODE;
GOTO LBRACE IF C1 = 0;
GOTO ERROR;*      SYNTAX ERROR IN PARAMETER LIST
IOD:
PPR:
COMMAND:
GOTO NOTYET;
GOTO NOTYET;
GOTO NOTYET;
*IF AR0 IS A LEGITIMATE ARRAY AND IS RELEASSABLE, FREE IT.
PRINTAR:
C1 ← AR0 AND MASK4;
C1 ← C1 EOR APCODE;
GOTO NEXTLINE IF C1 ≠ 0;
C1 ← AR0 EOR NULLCODE;
GOTO NEXTLINE IF C1 = 0;
*IT IS A LEGITIMATE ARRAY:
C1 ← NEXTLINE;*      RETURN FOR NONRELEASEBLE
C2 ← FREEIT;*
VMA ← AR1;
GOTO RELEASETEST;
FREEIT:
RETURN ← NEXTLINE;
VMA ← AR1;
GOTO RELEASEBLOCK;

*
*
*
PM:
VMA ← AL1;*POINTER TO VALUE
RETURN ← OTYPECHECK;
GOTO ACCESSPHANTUM;
*

```

```

*****
* N - D I M E N T I O N A L   I N D E X I N G   *
*****
*
* AL CONTAINS LA      : V
* AR CONTAINS SUBSCRIPT: S1
* OPR CONTAINS I AND HAS CALLED THIS ROUTINE
*
INDEX:      TURNON DYADICB;
            VMD ← AL0 AND MASK4;
            VMD ← VMD EOR APCODE;
            GOTO ERROR IF VMD # 0;
            VMD ← AL0 EOR NULLCODE;
            GOTO ERROR IF VMD = 0; * NON NULL ARRAY V
            CALL DEFER; * STACK OPR AR
            STACK ← STACK - 2;
            GO PUSH;
            VMD ← AL1;
            GO PUSH;
            VMD ← AL0;
            GO PUSH;
*ESTABLISH V TAIL :
            VMA ← AL1 + 1;
            GO READ;
            VMD ← AL1 + VMD;
            GO PUSH;
*ALL SET FOR THE LRPASS !
LRPASS:     C4 ← 0; * INDEXCOUNT
            VMA ← STACK + 3; * INITIALIZE
LRLOOP:     C4 ← C4 + 1; * INCREMENT INDEXCOUNT
            VMA ← VMA + 6; * EXAMINE PREVIOUS OPERATOR
            GO READ;
            C1 ← VMD EOR SEMICLNCODE;
            GOTO LRLOOP IF C1 = 0;
            C1 ← VMD EOR RBRACKETC;
            GOTO ERROR IF C1 # 0;
*ALL SUBSCRIPTS COUNTED.
*CHECK FOR SPECIAL CASE OF ONE SUBSCRIPT.
            C1 ← C4 EOR 1;
            GOTO CONFORM IF C1 # 0;
*ONLY S1:
            C1 ← AR0 EOR NULLCODE; * NULL?
            GOTO CONFORM IF C1 # 0;
            AR0 ← AL0;
            AR1 ← AL1;
            VMA ← AL1;
            GOTO RELVEC2; * OUT
*CHECK INDEXCOUNT = V'S #DIM:
CONFORM:    VMD ← AL0 AND KILL4;
            VMD ← C4 EOR VMD;
            GOTO ERROR IF VMD # 0;
*ENTER INDEXCOUNT AND SN POINTER IN STACK:
            VMD ← VMA - 2;

```

GO PUSH;  
VMD ← C4;  
GO PUSH;

\*

\*INITIALIZE THE INDEXING BLOCK AND CREATE R'S DESCRIPTOR

```

RLPASS:      D1 ← C4;*      ABSCISS;
              C1 ← 0;*      R'S # ELEMENTS
              C2 ← 0;*      R'S # DIMENSIONS
              C3 ← 0;*      INDEXER
              C4 ← 1;*      NEXTPROD
              PTR2 ← AL1;*   VPOINTER
              VMA ← STACK + 1;
              GO READ;
              PTR1 ← VMD + 6;*   RUNNING POINTER ON STACK
              D4 ← AL1 + HEADERSIZE;
              VMD ← AL0 AND KILL5;
              D4 ← D4 + VMD;*   VDIM(ABSCISS)-POINTER

```

\*THE RIPPLE OF SINISTER REPUTATION

\*IS USED HERE TO DETECT AN ARRAY OR MISSING INDEX

\*AND SCARE IT.

\*THUS KEEPING THIS RUNNING.

```

RLLOOP:      PTR1 ← PTR1 - 6;
              VMA ← PTR1;
              CALL LOADAR;*      SUBSCRIPT(ABSCISS)
*TEST NOW TYPE OF SUBSCRIPT(ABSCISS):
              GOTO NOTSCALAR IF AR0$0#0;* SCALAR
              VMD ← 0;*      NULL DIMENSION IS LEGAL
              GO PUSH;
              D4 ← D4 - 1;*      NEXT VDIM
              VMA ← D4;
              GO READ;
              MONITOR ← VMD;*      SAVE VDIM FOR A BRILLIANT FUTURE
              VMA ← PTR1 - 1;
              GO WRITE;*      ENTER VDIM(ABSCISS)
              VMD ← C4;
              VMA ← VMA - 1;
              GO WRITE;*      ENTER PROD
              VMD ← 1;*      DIGIT
              VMA ← VMA - 1;
              GO WRITE;
              C2 ← C2 + 1;
BOUNDS:      GOTO ERROR IF AR0$R = 0;* = 0 ?
              GOTO ERROR IF AR0$8 = 1;* < 0 ?
              CALL FIX;
              SPARED ← AR1 EOR -1;
              VMD ← MONITOR + SPARED P1;
              GOTO ERROR IF VMD$0 = 1;* VDIM < EI ?
*BOUNDS ARE OK. COMPUTE THE INC AND SPILL IT
*ARI CONTAINS THE FIXED EI.
              GOTO ENTER0 IF C3 = 0;*FIRST ELEMENT
              AR1 ← AR1 - 1;
              CALL FLOAT;*      E - 1
              AL0 ← AR0;
              AL1 ← AR1;
              AR0 ← 0;
              AR1 ← C4;
              CALL FLOAT;*      PROD

```

```

                                CALL FMUL;*
                                CALL FIX;*
                                INC = (E - 1) * PROD
                                ARI HAS FIXED INC
ENTER0:   VMD ← ARI;
ENTERINC: VMA ← VMA - 1;
                                GO WRITE;*
                                ENTER INC
*INCSUM = INCSUM + INC
                                C3 ← C3 + ARI;*
                                INDEXER
*DECREMENT ABSCISS:
                                D1 ← D1 - 1;
                                GOTO RLDONE IF D1 = 0;
*NESTPROD(C4) ← (NEXT)PROD * D(I) :
                                ARO ← 0;
                                ARI ← C4;
                                CALL FLOAT;*
                                PROD
                                AL0 ← ARO;
                                AL1 ← ARI;
                                ARO ← 0;
                                VMA ← VMA + 3;
                                GO READ;
                                ARI ← VMD;
                                CALL FMUL;
                                CALL FIX;
                                C4 ← ARI;*
                                NEXTPROD
                                GOTO RLLOOP;
*D1 = ABSCISS = 0 :
RLDONE:   GOTO RCREATE IF RIPPLEF = 1;
*GOTO CHARACTER CASE IF V IS CHARACTER.
                                C3 ← C3 L1;*
                                DOUBLE IT
                                C3 ← C3 EOR -1;
                                VMD ← C2;
                                VMA ← STACK + VMD;
                                VMA ← VMA + 2;
                                GO READ;
                                VMA ← C3 + VMD P1;*
                                POINTER TO ELEMENT
                                CALL LOADAR;
                                GOTO SCADEX;
*WE ASSUME THAT R IS SCALAR. OTHERWISE, MODIFY WORD COUNT.
RCREATE:  D1 ← C1;
                                D2 ← C2;
                                C1 ← C1 L1;*
                                DOUBLE # ELEMENTS
                                C1 ← C1 + HEADERSIZE;
                                VMD ← C2;*
                                R'S # DIMENSIONS
                                C1 ← C1 + VMD;*
                                TOTALSIZE
                                CALL GETBLOCK;
*MONITOR DOES NOT MATTER ANYMORE.
*SAVE POINTER TO T IN C2.
*SAVE RUNNING R POINTER.
                                MONITOR ← VMD;
*CREATE HEADER:
                                VMA ← C2;
                                GO READ;
                                VMD ← VMD OR D2;*
                                # DIMENSIONS
                                GO WRITE;
                                VMA ← C2 + 2;

```

```

      VMD ← D1;
      GO WRITE;*                # ELEMENTS
*NOW, TRANSFER [STACK,STACK+D2-1] TO [C2+4,C2+4+D2-1] :
      C1 ← D2;*                COUNT
      PTR1 ← STACK;
      PTR2 ← C2 + 4;
      CALL XFERLOOP;
      STACK ← STACK + D2;
      TURNOFF RIPPLEM;
      GOTO STOREIT;
*POINTER TO RESULT BLOCK MUST BE SET !!!!!!!!!!!
*
*
```

```

NOTSCALAR:      VMD ← AR0 EOR NULLCODE;
                  GOTO MISSINGDEX IF VMD = 0;
                  VMD ← AR0 AND MASK4;
                  VMD ← VMD EOR APCODE;
                  GOTO ERROR IF VMD ≠ 0;
*ARRAY SUBSCRIPT:
                  GOTO ERROR IF AR0$4=1;* MUST BE NUMERIC
*GET # ELEMENTS AND ADJUST C1.
*USE INTMULT:
*:::::::::::
                  AR0 ← AR0 AND KILL5;* # DIMENSIONS
                  AR1 ← AR1 + HEADERSIZE;
                  C2 ← C2 + AR0;
                  AR1 ← AR1 + AR0;* POINT TO LAST SDIM
*TRANSFER DIMENSIONS (BLOCK OF LENGTH AR0) UNTO THE STACK:
MOVEDIMS:      AR1 ← AR1 - 1;
                  VMA ← AR1;
                  GO READ;
                  LINK ← MOVEDIMS;
                  AR0 ← AR0 - 1;
                  GOTO PUSH IF AR0 ≠ 0;
DIMMOVED:      VMA ← VMA + 1;
                  GO READ;* # ELEMENTS
*GET NOW FIRST ELEMENT OF ARRAY SUBSCRIPT:
                  VMA ← VMA - 1;
                  GO READ;* SIZE
                  VMA ← VMA + VMD;
                  VMA ← VMA - 3;
                  CALL LOADAR;
                  TURNON RIPLEB;
                  VMA ← PTR1 - 4;
                  GOTO BOUNDS;

```

\*



```

*NULL HAS BEEN FOUND AS A SUBSCRIPT : MISSING INDEX;
MISSINGDEX:  C2 ← C2 + 1;*          R'S # DIMS
*STACK D1 , CALL INTMULT , POP D1.
*:::::::::::
                VMD ← MONITOR ;*          VDIM
                GO PUSH;*              NEW RDIM
*::::::::::: COMPUTE R'S # ELEMENTS
                TURNON RIPPLEB;
                VMD ← 0;*              INC
                VMA ← PTR1 - 4;*        INC POINTER
                GOTO ENTERINC;

*
*
*-----*
*  I N T M U L T   S U B R O U T I N E  *
*-----*
INTMULT:        RETURN ← D1;
                AR0 ← 0;
                AR1 ← VMD;
                CALL FLOAT;
                AL0 ← AR0;
                AL1 ← AR1;
                AR0 ← 0;
                AR1 ← C1;
                CALL FLOAT;
                CALL FMUL;
                CALL FIX;
                LINK ← D1;
                C1 ← AR1 J;

*=====*
*
*WE ASSUME NUMERIC FOR STORING:
STOREIT:        VMA ← STACK - 2;*      ACCESS V ELEMENT
                READIT;*              TAIL IN VMD;
                SPARED ← C3 L1;*      DOUBLE THE INDEXER
                VMA ← SPARED + VMD;
                GO READ;
                AR0 ← VMD;
                VMA ← VMA - 1;
                GO READ;
                AR1 ← VMD;*          V'S ELEMENT IS ACCESSED
                C1 ← C1 + 1;*      R'S # ELEMENTS
*ENTER THE ELEMENT IN R:
                MONITOR ← MONITOR + 2;
                VMA ← MONITOR;*      RUNNING R POINTER
                VMD ← AR0;
                GO WRITE;
                VMA ← VMA - 1;
                VMD ← AR1;
                GO WRITE;
                GOTO RIPPLETHRU IF RIPPLEF = 0;

*
*AR IS LOADED WITH LAST ELEMENT.

```

```

*LOAD IT WITH RESULT POINTER.
XDONE:      VMA ← C2;*      R POINTER LOCATION
            READIT;
            AR1 ← VMD;
            VMA ← VMD;
            GO READ;
            C1 ← VMD AND KILL4;
            AL0 ← C1 OR APCODE;
            AL1 ← VMD;
*ALLSET. RELEASE FLAGS.
SCADEX1:    TURNOFF RIPPLEM;
*RELEASE ALL S BLOCKS WHICH ARE RELEASABLE:
            GO POP;
            D1 ← VMD;*INDEX COUNT.
            STACK ← STACK + 2;
            GO POP;
            D2 ← VMD;*      V POINTER
CLEANSTACK: D1 ← D1 - 1;
            STACK ← STACK + 6;
*  SAVE AR (RESULT) !!!!
            CALL LOADAR;
            GOTO APTYPE IF AR0$0#0;
ENDTEST:    GOTO CLEANSTACK IF D1 # 0;
            VMA ← D2;*      V POINTER
            AR0 ← AL0;*      R POINTER
            AR1 ← AL1;
            SETSTATE1;
            GOTO RELVEC2;*      EXIT
*
SCADEX:      STACK ← STACK + 13;
            SETSTATE1;
            GOTO UNSTACK;
*
APTTYPE:     SPARED ← AR0 EOR NULLCODE;
            GOTO ENDTEST IF SPARED = 0;
            VMA ← AR1;
            C1 ← ENDTEST;
            C2 ← ENDTEST;
*CORRECTION: GOTO RELEASEBLOCK, THEN ENDTEST ! *:::::::::::
            GOTO RELEASETEST;
*

```

```

*RIPPLETHRU:
*-----*
*
* C1 IS R'S # CREATED ELEMENTS
* C2 IS R POINTER
* C3 IS INCSUM
* C4 IS WORKING REGISTER
* D1 IS ABSCISS
* PTR1 IS S POINTER
* MONITOR IS R RUNNING POINTER
* PTR2 IS ABSCISS.
*
*-----*
*
RIPPLETHRU:    C1 ← 0;
               C3 ← 0;*                INDEXER
               TURNON RIPPLEB;
*READ INDEX COUNT:
               VMA ← STACK;
               READIT;
               PTR2 ← VMD;
               VMA ← STACK - 1;
               READIT;
               PTR1 ← VMD + 6;*          CURRENT S POINTER
               PTR1 ← PTR1 - 6;
RIPPLELOOP:   VMA ← PTR1;
               CALL LOADAR;
               GOTO XARRAY IF AR0$0#0;
*SCALAR SUBSCRIPT - JUST READ INC.
READING:      VMA ← PTR1 - 4;
               GO READ;*                VMD HAS INC
               GOTO NUINDEXER;
XARRAY:       GOTO READING IF RIPPLEF = 0;
* RIPLE IS ON -- CYCLE THE ODOMETER.
*READ DIGIT OF ODOMETER:
               VMA ← PTR1 - 3;
               GO READ;
               SPARED ← VMD EOR -1;*    - DIGIT
               VMD ← VMD + 1;
               GO WRITE;
*OVERFLOW OF ODOMETER DIGIT ?- GET VDIM
               VMA ← PTR1 - 1;
               GO READ;
               VMD ← VMD + SPARED P1;
               GOTO ODOMOVER IF VMD$0 = 1;
               TURNOFF RIPPLEM;
NUDIGIT:      VMA ← PTR1 - 3;
               GO WRITE;
               SPARED ← AR0 EOR NULLCODE;
               GOTO MICASE IF SPARED = 0;
*TRUE ARRAY: ACCESS SI(DIGIT) = NEW EI
               FREE ← VMD L1;*          DOUBLE DIGIT
               VMA ← AR1 + 1;

```

```

READIT;
SPARED ← AR1 + VMD;*      TAIL
FREE ← FREE EOR -1;
VMA ← SPARED + FREE P1;
CALL LOADAL;
AR0 ← FLOATINGONE;
AR1 ← 0;
CALL FLSUB;

*AR HAS EI -1
COMPUTINC:    VMA ← PTR1 - 2 ;*      GET PRD;
              CALL LOADAL;
              CALL FMUL;
              CALL FIX;
              VMA ← PTR1 - 4;
              VMD ← AR1;
              GO WRITE;
NUINDEXER:   C3 ← VMD + C3;
              PTR2 ← PTR2 - 1;
              GOTO STOREIT IF PTR2 = 0;
              GOTO RIPPLELOOP;

*VMD HAS DIGIT.
MICASE:       AR0 ← 0;
              AR1 ← VMD - 1;
              CALL FLOAT;
              GOTO COMPUTINC;

*
UDOMOVER:     VMD ← 1;
              GOTO NUDIGIT;

*
*
*
```

```

***** OPERATORS FRONT END *****
=====
*
*AN OPERATOR HAS BEEN RECOGNIZED AS:
*   MONADIC (MONOPER)
*   OR DYADIC (BINOPER)
*THE OPERATOR IS IN OPR.
*THE ARGUMENTS ARE IN AL AND AR
*IN THE CASE OF A MONADIC OPERATOR, THE "NEXT" OPERATOR IS IN C4.
*THE RESULT IS LEFT IN AR.
*THE STATE WILL BE 1
*C4 MUST NOT BE MODIFIED BY THE OPERATOR ROUTINES.
*THE LOOPANG FOR A MONADIC OPERATION ON IN ARRAY
*IS HANDLED BY THE DYADIC CASE WITH A 0-0 LOADED IN AL.
*THE 2 CASES CAN BE RECOGNIZED LATER BY THE DYADIC FLAG (0 OR 1)
*
MONOPER:      C1  ← OPR EOR LBRACKETCODE;*      [ IS SPECIAL.
               GOTO OPINDEX IF C1 = 0;
               C1  ← OPR EOR DOTCODE;
               GOTO DOTOPER IF C1 = 0;*          DOT IS SPECIAL
               TURNOFF DYADICM;*                SET MONADIC MODE
               AL0 ← 0;
               GOTO OPTYPETEST;*                VERIFY SYNTAX
*
*
BINOPER:      C1  ← OPR EOR LBRACKETCODE;*      [ IS SPECIAL
               GOTO INDEX IF C1 = 0;
               C1  ← OPR AND MASK4;
               C1  ← C1 EOR FCCODE;*            PENDING FUNCTION CALL
               GOTO BINFC IF C1 = 0;
               TURNON DYADICB;*                SET DYADIC MODE
*NOTE : A MIXEDOP DOES NOT OPERATE ELEMENT BY ELEMENT AND
*NECESSITATES A SPECIAL FRONT END.
OPTYPETEST:   GOTO OPERATE IF MIXEDOPBIT=1;* TEST FOR MIXEDOPS
               GOTO XXXSCA IF AR0$0=0;*        AR SCALAR ?
               C1  ← AR0 AND MASK4;
               C1  ← C1 EOR APCODE;
               GOTO ERROR IF C1 # 0;*          ILLEGAL ARGUMENT TYPE
               C1  ← AR0 EOR NULLCODE;*        NULL ?
               GOTO NULL IF C1 = 0;
XXXVEC:       GOTO SCAVEC IF AL0$0 = 0;*        AR ARRAY ?
               C1  ← AL0 AND MASK4;
               C1  ← C1 EOR APCODE;
               GOTO ERROR IF C1 # 0;*          ILLEGAL ARGUMENT(LEFT)
               C1  ← AL0 EOR NULLCODE;*        NULL ?
               GOTO NULL IF C1 = 0;
               GOTO VECVEC;
*
XXXSCA:       GOTO SCASCA IF AL0$0 = 0;
               C1  ← AL0 AND MASK4;
               C1  ← C1 EOR APCODE;
               GOTO ERROR IF C1 # 0;*          ILLEGAL LEFT ARG
               C1  ← AL0 EOR NULLCODE;

```

```
GOTO NULL IF C1 = 0;  
GOTO VECSCA;
```

```
*  
*
```

\*TEST C4 FOR INDEXABLE OPERATOR:

```
OPINDEX:      GOTO ERROR IF C4$9=0;*   ILLEGAL USE OF [
              C1  ← C4 EOR BACKSLASH;
              GOTO XSLASH IF C1 = 0;
              C1  ← C4 EOR SLASHCODE;
              GOTO SLASH IF C1 = 0;
              C1  ← C4 EOR TRANSPCODE;
              GOTO XPAND IF C1 = 0;
              GOTO ERROR;*              ILLEGAL INDEXED OPERATOR
```

\*

\*

\*OPR CONTAINS A DYADIC FUNCTION CALL

\*THE FIRST ARGUMENT HAS BEEN STACKED AN AD IS IN AR ALSO

\*THIS ROUTINE IS ENTERED VIA LPAREN OR

\*DIRECTLY UPON DETECTION OF A SIMPLE LEFT ARGUMENT

\*OR BY BACKUP AFTER COMPLETION OF AN OPERATION ON THE LEFT

```
BINFC:      AR0 ← AL0;*              TRANSFER AL TO AR FOR DEFER
```

```
            AR1 ← AL1;
```

```
            CALL DEFER;
```

```
            STACK ← STACK + 1;*      MOVE PTR BACK, SINCE OPR
```

```
            AL0 ← OPR;*  RESTORE FC
```

\*

NEEDS NOT BE STACKED BY DEFER

```
            AL0 ← OPR;*RESTORE FC
```

```
            GOTO NUFRAME;*          VMD CONTAINS OPR !
```

\*

```

SCASCA:      \   TURNOFF CHARM;*          SCALAR MODE WHEN OFF
              TURNON SCASCAB;
OPERATE:     RETURN ← OPRETURN;
DOIT:        GOTO MONADIC IF DYADICF = 0;
DYADIC:      GOTO BINGO IF OPR$7 = 0;*    LOGICAL GROUP?
              SPARED ← AR0;
              VMD ← AR1;
              GO LOGICALVERIFY;*          VERIFY AR
              SPARED ← AL0;
              VMD ← AL1;
              GO LOGICALVERIFY;*          VERIFY AL IS LOGICAL
BINGO:       LINK ← OPR AND KILL10;*      EXTRACT OPCODE
              LINK ← LINK L1;*           DOUBLE IT
              EXECUTE BINOPTABLE IX;

*
ORIGIN 4300B;
BINOPTABLE:  GOTO FLADD;*                +          00
              GOTO FLSUB;*               -          01
              GOTO FMUL;*                $TIM       02
              GOTO FDIV;*                $DIV       03
              GOTO MAX;*                 $TOP       04
              GOTO MIN;*                 $BOT       05
              GOTO RES;*                 $BAR       06
              GOTO BAND;*                $AND       07
              GOTO BOR;*                 $OR        10
              GOTO ERROR;*               $NOT       11
              GOTO NAND;*                $NAND      12
              GOTO NOR;*                 $NOR       13
              GOTO LESS;*                <          14
              GOTO LEQ;*                 $LE        15
              GOTO GEQ;*                 $GE        16
              GOTO GREATER;*             >          17
              GOTO EQUAL;*               =          20
              GOTO NOTEQUAL;*            #          21
              GOTO ERROR;*               NOTHING    22
              GOTO INDEXOF;*             $IOTA     23
              GOTO RESHAPE;*             $RHO      24
              GOTO CATENATE;*            ,          25
              GOTO COMPRESS;*            /          26
              GOTO EXPAND;*              \          27
              GOTO POWER;*               *          30

```

\*



```

MONADIC:      GOTO MONGO IF OPR$7 = 0;*      LOGICAL TEST?
              SPARED  ← AR0;
              VMD     ← AR1;
              GO LOGICALVERIFY;*              VERIFY AR
MONGO:        LINK ← OPR AND KILL10;*          EXTRACT OPCODE
              LINK ← LINK L1;
              EXECUTE MONOPTABLE IX;

```

✱

ORIGIN 4500B;

\*ERRORS HERE, UNLESS NOTED OTHERWISE ARE

\*ILLEGAL USE OF A DYADIC OPERATOR IN A MONADIC CONTEXT.

MONOPTABLE:	GOTO OUT;* *	+	00
	GOTO UMINUS;* *	-	01
	GOTO SIGNUM;* *	\$TIM	02
	GOTO RECIPROCAL;* *	\$DIV	03
	GOTO CEILING;* *	\$TOP	04
	GOTO FLOOR;* *	\$BOT	05
	GOTO MAGNITUDE;* *	\$BAR	06
	GOTO ERROR;* *	\$AND	07
	GOTO ERROR;* *	\$OR	10
	GOTO NOT;* *	\$NOT	11
	GOTO ERROR;* *	\$NAND	12
	GOTO ERROR;* *	\$NOR	13
	GOTO ERROR;* *	<	14
	GOTO ERROR;* *	\$LE	15
	GOTO ERROR;* *	\$GE	16
	GOTO ERROR;* *	>	17
	GOTO ERROR;* *	=	20
	GOTO ERROR;* *	#	21
	GOTO ERROR;* *		22
	GOTO IOTA;* *	\$IOTA	23
	GOTO SIZE;* *	\$RHO	24
	GOTO RAVEL;* *	,	25
	GOTO ERROR;* *	/	26
	GOTO ERROR;* *	\	27
	GOTO EXPO;* *	*	30

\* \* \* \* \*

```

* C4 IS THE LA RUNNING PTR
* PTR1 IS THE RA RUNNING PTR
* PTR2 IS THE RESULT
*
*****
*
VECVEC:      VMD  + AR0 AND KILL5;
              GO PUSH;*
              TURNON VECVEC;
              C1   + AL0 EOR AR0;*
              GOTO ERROR IF C1 # 0;*
              C2   + AR0 AND KILL5;*
              PTR1 + AR1 + HEADERSIZE;*
              C4   + AL1 + HEADERSIZE;*
*CHECK NOW # ELEMENTS FOR EACH DIMENSION.
DIMCHECK:    VMA  + PTR1;*
              GO READ;
              C1   + VMD;*
              VMA  + C4;*
              GO READ;
              C1   + C1 EOR VMD;*
              GOTO ERROR IF C1 # 0;*
              C2   + C2 - 1;*
              PTR1 + PTR1 + 1;
              C4   + C4 + 1;
              GOTO DIMCHECK IF C2 # 0;*
*THEY ARE CONFORMABLE.
              VMA  + AR1 + 2;*
              GO READ;
              D1   + VMD;*
              VMA  + AR1 + 1;*
              GO READ;
              PTR1 + AR1 + VMD;*
              VMA  + AL1 + 1;*
              GO READ;
              C4   + AL1 + VMD;
*PTR1 AND C4 ARE NOW SET
              GOTO CHARCHAR IF AR0$4 = 1;*
              CALL INTSTB;*
*
SCALOOP:     CALL GETLSCALAR;
              CALL GETRSCALAR;
              CALL DOIT;
*ATTENTION: C4,PTR1,PTR2,FREE MUST BE PRESERVED BY THESE
*            ROUTINES.      BEWARE !!!!
              RETURN + SCALOOP;
              GOTO SVLOOP;
*

```

SAVE # DIMS  
 OK FOR TYPE AND # DIMS?  
 ARRAYS NOT CONFORMABLE  
 # DIMS  
 RA'S #ELEM  
 LA'S #  
 # ELEM FOR DIM I  
 RA'S # ELEM  
 LA'S  
 COMPARE FOR THE DIM  
 RANK ERROR.SEEC2 FOR DIM  
 # DIMENSIONS TO BE CHECKED  
 ITERATE?  
 POINT TO # ELEMENTS  
 COUNTER IS # ELEMENTS  
 SIZE  
 POINT BENEATH BASE  
 SAME FOR LA  
 RA IS CHAR  
 ALLOCATE BLOCK FOR RESULT  
 THIS SETS PTR2

```

*ITERATION LOOP FOR A SCALAR OPERATOR OPERATING ON CHARACTER ARRAYS
*TYPE HAS ALREADY BEEN CHECKED
*THE ONLY LEGAL APL\360 OPERATORS ARE = AND #. MORE COULD BE ADDED.
CHARCHAR:      TURNON CHARB;*      CHARACTER MODE
               C1  ← OPR EOR EQUALCODE;*      = ?
               GOTO CORRECTOPER IF C1 = 0;
               C1  ← OPR EOR NEQUALCODE;*      # ?
               GOTO ERROR IF C1 # 0;*      ILLEGAL SCALOP 'OPR'
                                           FOR CHAR VECTORS
*
*SINCE IN THIS CASE, RESULT IS 4* SIZE OF OPERANDS , A NEW BLOCK
*MUST BE ALLOCATED .
* SIZE(NEWBLOCK)=SIZE(HEADER) + 2*(#ELEMENTS) + # DIMENSIONS.
*COMPUTE ITS SIZE.
*FREE CONTAINS THE # ELEMENTS
CORRECTOPER:   C1  ← D1 L1;* 2*(ELEMENTS)  VERIFY THAT D1 IS OK !!
               C1  ← C1 + HEADERSIZE;
               SPARED ← AR0 AND KILL5;*      # DIM
               C1  ← C1 + SPARED;*      SIZE(BLOCK)
               CALL GETBLOCK;
               PTR2 ← VMD;*      RESULT PTR SET
               C4  ← C4 - 1;*      LA'S FIRST WORD
CHARLOOP:      CALL GETLCHARACTER;
MONCHARLOOP:   CALL GETRCHARACTER;
               CALL DOIT;
               RETURN ← CHARLOOP;
               GOTO SVLOOP;
*

```

```

VECSA:      GOTO ERROR IF AL0$4 = 1;*      LA MAY NOT BE CHAR
            GO SWAPREGISTERS;
SCAVEC:     VMD ← AR0 AND KILL5;*          #DIMS
            GO PUSH;
            VMA ← AR1 + 2;*                # ELEMS
            GO READ;
            D1 ← VMD;*                      COUNTER = # ELEM
            VMA ← AR1 + 1;*                  SIZE
            GO READ;
            PTR1 ← AR1 + VMD;
            CALL INTESTB;*
            GOTO SCACHAR IF AR0$4 = 1;*    ALLOCATE BLOCK FOR RESULT
            SAVEAL;                        CHAR ARRAY ?

*SCALAR-SCALAR:
            GOTO SCAVELOOP;
*SCALAR-CHARACTER:
SCACHAR:    GOTO ERROR IF DYADICF=1;*      OPERATOR MUST BE MONADIC
            GOTO MONCHARLOOP;
*
SCAVELOOP:  CALL GETRSCALAR;
            CALL DOIT;
            RETURN ← SCAVELOOP;
            GOTO SVLOOP;
*
```

\*\*\*\*\*

\*ROUTINES\*

\*\*\*\*\*

```

*-----
GETRSCALAR:  PTR1  ← PTR1 - 2;* MOVE UP POINTER IN RA
              VMA   ← PTR1;
              GO READ;
              AR0    ← VMD;
              VMA    ← VMA + 1;
              GO READ;
              LINK   ← RETURN;
              AR1    ← VMD J;

```

```

*-----
GETLSCALAR:  C4     ← C4 - 2;
              VMA   ← C4;
              GO READ;
              AL0    ← VMD;
              VMA    ← VMA + 1;
              GO READ;
              LINK   ← RETURN;
              AL1    ← VMD J;

```

```

*-----
GETRCHARACTER: VMA   ← PTR1;
                GO READ;
                LINK  ← RETURN;
                GOTO REVENCOUNTER IF D1$15=0;
                PTR1  ← PTR1 - 1;
                AR0    ← VMD R8,J;

```

```

*
REVENCOUNTER:  AR0    ← VMD AND KILL8;*      EXTRACT RIGHT HALF
                ZERO J;

```

```

*-----
GETLCHARACTER: VMA   ← C4;
                GO READ;
                GOTO LEVENCOUNTER IF D1$15=0;
                AL0    ← VMD R8;*      GET LEFT HALF FOR ODD COUNTER
                C4     ← C4 -1;*      POINT TO NEXT UP
                EXITTORETURN;

```

```

*
LEVENCOUNTER:  AL0    ← VMD AND KILL8;*      EXTRACT RIGHT HALF
                EXITTORETURN;

```

\*

```

*-----
*DANGER: RETURN GETS USED AT THE END. WATCH IT, GREEDBIRD !
*DON'T USE IT IN INTERMEDIATE ROUTINES.!
*STORAGE ALLOCATION FOR T = A OP B.
*****
*
*   AR1 POINTS TO A
*   AL1 POINTS TO B
*   POINTER BENEATH RESULT T IS RETURNED IN PTR2.
*   HEADER WILL HAVE BEEN CREATED.
*
INTESTB:      VMA   ← AR1;*                SET AP
              C1    ← CHECKA;*            RETURN FOR NONRELEASABLE
              C2    ← SETRESULTPTR;*
              GOTO RELEASETEST;*          IS B RELEASABLE ?
*B IS RELEASABLE
*ALLOCATE IT FOR RESULT. NO CHANGE TO HEADER NEEDED.
SETRESULTPTR: C2 ← VMA - 3;
              VMA   ← C2 + 1;*            SIZE
              GO READ;
              LINK   ← RETURN;
              PTR2   ← C2 + VMD J;*        RESULT PTR SET
*
*
*
*B IS NOT RELEASABLE.
*
*IS A RELEASABLE ?
*
CHECKA:       C1     ← AL0 AND MASK4;
              C1     ← C1 EOR APCODE;
              GOTO NONEFREE IF C1 ≠ 0;
              C1     ← C1 EOR NULLCODE;
              GOTO NONEFREE IF C1 = 0;
              VMA    ← AL1;
              C1     ← NONEFREE;*         IF NONRELEASABLE
              C2     ← SETRESULTPTR;*     IF RELEASABLE
              GOTO RELEASETEST;
*
*A IS NOT RELEASABLE.
*SET BLOCK SIZE IN C1.
*
NONEFREE:     VMA    ← AR1 + 1;*          SIZE
              GO READ;
              C1     ← VMD;
              C3     ← RETURN;*          SAVE RETURN;
              CALL GETBLOCK;
*NOW TRANSFER VECTOR HEAD.
*C2 POINTS TO WORD 0.
*VMD POINTS BENEATH NEW BLOCK.
              PTR2   ← VMD;
*STACK RESULT AP ????????? AS IN SETRESULTPTR.
              C1     ← AR1;

```

```
SPARED ← AR0 AND KILLS;*          #DIM  
COUNTER ← SPARED + HEADERSIZE;  
HEADXFER:  
VMA ← C1;  
GO READ;  
VMA ← C2;  
GO WRITE;  
C1 ← C1 + 1;  
LINK ← HEADXFER;  
C2 ← C2 P1,D,J;  
LINK ← C3;  
ZERO J;  
  
*
```

```

*-----
*SUCCESSFUL EXIT THRU C2 (RETURN IS NEEDED BY INTESTB)
*UNSUCCESSFUL EXIT .. C1.
*VMA MUST POINT TO WORD 0 OF ARRAY.
*THIS DOES NOT DISTURB ANY REGISTER.
*
*
RELEASESETEST:  GO READ;*                                GET WORD 0 OF ARRAY
                GOTO NONRELEASABLE IF VMD$1 = 0;* RELEASABLE?
                VMA ← VMA + 3;*                          F IS ON - GET DEF FIELD
                GO READ;
                VMD ← VMD EOR -1;*                        - SAVEDSTACK
                VMD ← STACK + VMD P1;*                   STACK - SAVEDSTACK
                GOTO NONRELEASABLE IF VMD$0 = 1;*         <0?
RELEASEABLE:    LINK ← C2;
                VMD ← 0;
                GOTO WRITE;*                                ZERO SAVEDSTACK
*
NONRELEASABLE: LINK ← C1;
                ZERO J;
*
*
*-----
*
* C1 CONTAINS THE # OF WORDS TO BE ALLOCATED
*THIS TEMPORARY ROUTINE COULD BECOME A FIRST FIT ALGORITHM.
* C4 MAY NOT BE USED : HOLDS PREVIOUS RETURN.
*
GETBLOCK:       VMA ← BLOP;*                                GER BLOP FROM GFRZ
                GO READ;
                C2 ← VMD;
                VMA ← C2;
                VMD ← 0;
                GO WRITE;
                VMA ← C2 + 1;
                VMD ← C1;
                GO WRITE;
                VMA ← C2 + 3;
                VMD ← 0;
                GO WRITE;
                VMD ← C2;
                VMD ← C1 + VMD;
                LINK ← RETURN;
                VMA ← BLOP;
                GOTO WRITE;*                                SAVE NEW BLOP
*
*RETURNS WITH VMD POINTING BELOW NEW BLOCK.
*AND C2 TO WORD 0.
*

```



```

*-----
* SCALAR VECTOR LOOP
* PTR2 POINTS TO RESULT
* AL GETS DESTROYED BY SOME OPERATOR ROUTINES
*
SVLOOP:      PTR2 ← PTR2 - 2; *           MOVE UP RESULT PTR
             VMA  ← PTR2;
             VMD  ← AR0; *           ENTER AR
             GO WRITE;
             VMA  ← VMA + 1;
             VMD  ← AR1;
             GO WRITE; *
*RESTORE THE AL DESTROYED BY FPP.
             RESTOREAL;
             D1 ← D1 - 1; *           DECREMENT COUNTER
             GOTO OPRETURN IF D1 = 0;
             EXITTORETURN;

*
*
*-----
*

```

```

*-----
* T = A OP B *
*-----*
*
OPRETURN:      SETSTATE1;
                GOTO NOMOREARRAY IF SCASCF = 1;
*IT WAS AN ARRAY OPERATION.
                GO POP;*
                FREE ← VMD + HEADERSIZE;      GET DIM# PREVIOUSLY SAVED
                AR0 ← VMD + APCODE;
                FREE ← FREE EOR -1;
                AR1 ← PTR2 + FREE P1;*        RESULT POINTER
                VMA ← AR1;*                  ACCESS WORD 0 OF RESULT
                GO READ;
                VMD ← VMD OR FLOATINGB;* TURN IT ON !
                GO WRITE;
RELEASEVEC:    VMA ← PTR1 + FREE P1;*        RA
RELVEC2:       C1 ← SECONDARRAY;
                C2 ← **4;
                GOTO RELEASETEST;
*IT IS RELEASABLE. VMA POINTS TO DEFERMENT FIELD.
                VMA ← VMA - 3;
                CALL RELEASEBLOCK;
SECONDARRAY:   GOTO NOMOREARRAY IF VECVECF=0;
*THERE WERE TWO ARRAYS:
                TURNOFF VECVECM;
                VMA ← C4 + FREE P1;
                GOTO RELVEC2;
*NO ARRAYS WERE INVOLVED.
NOMOREARRAY:   TURNOFF OPMASK;
                GOTO WASMONADIC IF DYADICF = 0;
                TURNOFF DYADICM;
                GOTO PARSENEXT;
WASMONADIC:    VMD ← C4;*
                GOTO OPERATOR;                RESTORE THE NEXT OPERATOR
*

```

## \*\*\*\*\* D Y A D I C O P E R A T O R S \*\*\*\*\*

\*

\* FLOATING POINT PACKAGE:

\* SPARED IS ACC (ACCUMULATOR)

\* AR0 A

\* AR1 AQ

\* FREE AEXP

\* AL0 B

\* AL1 BQ

\* VMD BEXP

\*

\* PRIOR TO ANY DYADIC OPERATION, AL IS COPIED IN C1 C2.

\*

\* C1 AND C2 ARE UNAVAILABLE.

\* AL IS AVAILABLE

\* C4 IS NOT : LA RUNNING

\* PTR1 NOT : RA

\* PTR2 NOT : RESULT

\*

\*\*\*\*\*)

\*\*\*\*\*

\*\*\*\*\*

\*FLOAT\* INTEGER TO FLOATING NOTATION

\*\*\*\*\*

FLOAT: FREE ← 23;\*

INTEGER HAS EXPONENT 23

GOTO NORM;

\*\*\*\*\*

\*

\*\*\*\*\*

\*FIX\* FLOATING TO INTEGER

\*\*\*\*\*

\*INTEGER IS LEFT IN AR0 AR1 , RIGHT JUSTIFIED

\* NO FUZZ USED. IT TRUNCATES.

\*

FIX: GOTO FZERO IF AR0\$1=1;\*EXPONENT NEGATIVE?

FREE ← AR0 R8;\*SET FREE TO EXPONENT OF A

AR0 SE;\*EXTEND SIGN OF HIGH ORDER

FREE ← FREE EOR -1;

COUNTER ← FREE + 24;\*SET COUNTER TO 23-AEXP

GOTO FOVER IF COUNTER\$8=1;\*OVERFLOW IS FREE GT 23

GOTO OUT IF COUNTER\$R=0;\*NO SHIFT FREE = 23

ZERO ← AR0 + 200B;\*SET CARRY FOR SIGN PROP

LINK ← \* + 2;

AR0 ← AR0 OR ZERO AV,SO;

AR1 ← AR1 OR ZERO SI,R1,D,J;

GOTO OUT;

FZERO: AR1 ← ZERO;

AR0 ← ZERO;

GOTO OUT;

FOVER: TURNON INTOVERB;

GOTO ERROR;

\*

\*

```

*****
*FLSUB*
*****
*      FLOATING SUBTRACTION  (A,AQ SUBTRACTED FROM B,BQ TO A,AQ)
*
FLSUB:      GO BRKUPA;
            GO BRKUPB;
LINK ← ATEST;
GOTO COMPA;

*
*****
*COMPB*
*****
*      NEGATE B
*
COMPB:      AL1 ← AL1 EOR -1;
            AL0 ← AL0 EOR -1;
            AL1 ← AL1 + ZERO P1;
            AL0 ← AL0 + ZERO C1;
            GOTO OUT1 IF AL0$0=1;*RESULT NEGATIVE?
            GOTO OUT1 IF AL0$8=0;*FRACTION OVERFLOW?
            VMD ← VMD + 1;
            AL0 ← AL0 R1,S0;
            AL1 ← AL1 R1,S1,J;

*****
*NORM*
*****
*
*      NORMALIZE      (COEF IN A,AQ. EXP IN AEXP)
*      NORMALIZE      EXP IN FREE
*
NORM:      GOTO NRM IF AR1$W#0;*TEST FOR COEF ZERO
            GOTO OUT IF AR0$R=0;
NRM:      GOTO NNEG IF AR0$8=1;*NEGATIVE?
            GOTO EXITN IF AR0$9=1;*MSB = 1 FOR POS NUMBER?
SAME:      AR1 L1,S0;
            AR0 L1,S1;
            FREE ← FREE - 1;
            GOTO NRM;
NNEG:      GOTO EXITN IF AR0$9=0;*MSB = 0 FOR NEG NUMBER?
            GOTO SAME;
EXITN:      SPARED ← FREE + 64;*TEST FOR EXP UNDERFLOW
            GOTO UNDERF IF SPARED$0=1;
            SPARED ← FREE - 64;*TEST FOR EXP OVERFLOW
            GOTO OVERF IF SPARED$0=0;
NPACK:      FREE L8;*SHIFT OVER EXP
            FREE ← FREE AND 77600B;
            AR0 ← AR0 AND 377B;
            AR0 ← AR0 OR FREE;
            GOTO OUT;
UNDERF:      TURNON EXPUNDERB;
            GOTO OUT;
OVERF:      TURNON EXPOVERB;

```

GOTO OUT;

\*

```

*****
*FLADD*
*****
*
*      FLOATING ADD A,AQ PLUS B,BQ TO A,AQ.
*
FLADD:  LINK ← * + 10;
BRKUPA:  FREE ← AR0 R8;*EXPONENT TO AEXP
        SKIP1 IF FREE$9=0;
        FREE ← FREE OR 177600B;*SIGN EXTEND EXPONENT.
        AR0 ← AR0 SE,J;*HIGH ORDER
        LINK ← ATEST;
BRKUPB:  VMD ← AL0 R8;
        SKIP1 IF VMD$9=0;
        VMD ← VMD OR 177600B;
        AL0 ← AL0 SE,J;
ATEST:   GOTO NORM IF AL0$W=0;*          B EQUAL TO 0 ?
        GOTO AZERO IF AR0$W=0;*A EQUAL TO ZERO?
        FLAGS ← FLAGS AND 177770B;
        SPARED ← FREE EOR -1;
        COUNTER ← VMD + SPARED P1;*SET COUNTER TO VMD - AEXP
        GOTO ADDEM IF COUNTER$R=0;*EXPONENTS EQUAL?
        GOTO ABIG IF COUNTER$8=1;*DIFFERENCE NEGATIVE
        ZERO ← AR0 + 200B;*SET CARRY
        GO ASHFT;*SKIP AROUND FIRST TIME
        SKIP1 IF COUNTER$2=0;*NO SHIFT OUT?
        FLAGS ← FLAGS OR 1;*STICKY BIT
ASHFT:   AR0 ← AR0 OR ZERO AV,SO;
        AR1 ← AR1 OR ZERO R1,SI,SO,D,J;
        FREE ← VMD;
        GOTO ADDEM;*GO ADD EXPONENTS NOW EQUAL
ABIG:    SPARED ← VMD EOR -1;
        COUNTER ← FREE + SPARED P1;*SET COUNTER TO FREE - BEXP
        ZERO ← AL0 + 200B;*SET CARRY
        GO BSHFT;
        SKIP1 IF COUNTER$2=0;
        FLAGS ← FLAGS OR 1;*STICKY BIT
BSHFT:   AL0 ← AL0 OR ZERO AV,SO;
        AL1 ← AL1 OR ZERO R1,SI,SO,D,J;
ADDEM:   AR1 ← AL1 + AR1;
        AR0 ← AL0 + AR0 CI;
ADONE:   LINK ← ROUND;
AOVER:   GOTO APOS IF AR0$7=0;*NUMBER POSITIVE?
        GOTO SHFT IF AR0$8=0;
ABACK:   ZERO ← J;
APOS:    GOTO ABACK IF AR0$8=0;*TRAP FOR STOLENFILE
SHFT:    SKIP1 IF COUNTER$2=0;*LSOB GOES THRU STICKY BIT
        FLAGS ← FLAGS OR 1;*STICKY BIT
        FREE ← FREE + 1;
        AR0 ← R1,SO;
        AR1 ← R1,SI,SO,J;
ROUND:   GOTO NORM IF COUNTER$2=0;*LSOB = 0?
        GOTO UP IF FLAG$15=0;*STICKY BIT = 1?

```

```
      GOTO NORM IF AR1$15=0;*ROUND UP FOR ADD NUMBERS
UP:    AR1      P1;
      AR0      CI;
      LINK ← NORM;
      GOTO AOVER;
AZERO: AR0 ← AL0;
      AR1 ← AL1;
      FREE ← VMD;
      GOTO NORM;
*
*
```

```

*****
*FMUL*
*****
*
*      FLOATING MULTIPLY A,AQ TIMES B,BQ INTO A,AQ.
*
FMUL:   GOTO OUT IF AR0$R=0;*A EQUAL ZERO
        GOTO AZERO IF AL0$R=0;*B EQUAL ZERO?
        GO BRKUPA;
        GO BRKUPB;
        GOTO MULT IF AL0$R=0;*B NEGATIVE?
        GO COMPA;
        LINK ← MULT;
COMPA:  AR1 ← AR1 EOR -1;
        AR0 ← AR0 EOR -1;
        AR1 ← AR1 + ZERO P1;
        AR0 ← AR0 + ZERO CI;
        GOTO AOVER;
MULT:   FREE ← VMD + FREE;*ADD EXPONENTS.
        FLAGS ← FLAGS AND 177770B;*CLEAR FLAGS
        COUNTER ← 23;*SET COUNTER
        VMD ← AR1;
        SPARED ← AR0;
        AR0 ← ZERO;
        AR1 ← ZERO;
        GO SHFB;
        SKIP1 IF COUNTER$2=0;
        FLAGS ← FLAGS OR 1;*STICKY BIT
SHFB:   AL0 ← AL0 R1,S0;*SHIFT B,BQ FOR NEXT BIT
        AL1 ← AL1 R1,S0,SI;
        AR1 ← VMD * AR1;
        AR0 ← SPARED * AR0 CI,R1,S0;
        AR1 ← AR1 R1,SI,S0,D,J;
        GOTO ROUND;
*

```



\*\*\*\*\*  
 \*FDIV\*  
 \*\*\*\*\*

```

*
*      FLOATING DIVIDE  (B,BQ DIVIDED BY A,AQ INTO A,AQ)
*
FDIV:  SKIP2 IF AR0$R#0;*DIVIDE BY ZERO?
        FLAGS ← FLAGS OR 6;
        GOTO ERROR;
        GO BRKUPA;
            GO BRKUPB;
        FLAGS ← FLAGS AND 177770B;
        GOTO CKA IF AL0$8=0;*B POSITIVE
        GO COMPB;
        FLAGS ← FLAGS OR 2;
CKA:   GOTO BDIV IF AR0$8=1;*A NEG
        GO COMPA;
        FLAGS ← FLAGS EOR 2;*COMPLEMENT FLAGS$14
BDIV:  COUNTER ← 24;
        FREE ← FREE EOR -1;
        FREE ← VMD + FREE P1;
        VMD ← ZERO;
        SPARED ← ZERO;
        LINK ← * + 2;
        ZERO ← AL1 + AR1;*TEST DIVIDE LOW ORDER
        ZERO ← AL0 + AR0 SO,CI;*TEST DIVIDE HIGH ORDER
        GOTO NODIV IF COUNTER$2=1;*RESULT NEGATIVE SKIP
        AL1 ← AL1 + AR1;
        AL0 ← AL0 + AR0 CI;*SUBTRACT
NODIV: VMD ← VMD + ZERO L1,CI,SO;*SHIFT ANSWER
        SPARED ← SPARED + ZERO L1,SI;
        AL1 ← AL1 L1,SO;*SHIFT DIVIDEND
        AL0 ← AL0 L1,SI,D,J;
        AR0 ← SPARED R1,SO;
        AR1 ← VMD R1,SI,SO;
        GOTO ADONE IF FLAGS$14=1;
        LINK ← ADONE;
        GOTO COMPA;

```

\*

```

*****
*04:  MAXIMUM  *
*****
MAX:      SAVEAL;
          SAVEAR;
          MONITOR ← RETURN;
          CALL FLSUB;
          LINK ← MONITOR;
          SKIP2 IF AR0$8 = 1;
          CROSSRESTORE J;
          RESTOREAR J;

*
*****
*05:  MINIMUM  :
*****
*
MIN:      SAVEAR;*                AL IS ALREADY SAVED
          SAVEAL;
          MONITOR ← RETURN;
          CALL FLSUB;
          LINK ← MONITOR;
          SKIP2 IF AR0$8=0;
          CROSSRESTORE J;
          RESTOREAR J;

*
*****
*06:  RESIDUE  *
*****
*
RES:      A $BAR B
          SKIP2 IF AL0$R#0;*        LEFT ARG # 0 ?
          GOTO ERROR IF AR0$8=1;*    A = 0 , B < 0
          GOTO OUT;*                A = 0 , B > 0 RESULT B
          D2 ← RETURN;
          SAVEAL;
          SAVEAR;
          GOTO SAVEB IF AL0$8=0;*    A >= 0 ?
          AR0 ← AL0;
          AR1 ← AL1;
          CALL UMINUS;
          AL0 ← AR0;
          AL1 ← AR1;
SAVEB:    GO SWAPREGISTERS;
          CALL FDIV;
          CALL FLOOR;
          RESTOREAL;*              ABS B
          CALL FMUL;
          AL0 ← C3;
          AL1 ← VMA;* RESTORE B
          RETURN ← D2;
          GOTO FLSUB;

*
*****
*07:  AND      *

```

\*\*\*\*\*

\*

```
BAND:          LINK ← RETURN;
                GOTO 0 IF AR0$R=0 IX;*      B = 0 ?
                GOTO 0 IF AL0$R#0 IX;*      B = 1. A = 1 ?
                AR0 ← ZERO J;*              SET RESULT 0.
```

\*

\*\*\*\*\*

\*10: OR \*

\*\*\*\*\*

\*

```
BOR:          LINK ← RETURN;
                GOTO 0 IF AR0$R#0 IX;*      EXIT IF B=1
                GOTO 0 IF AL0$R=0 IX;*      B=0.A=0 ?
                AR0 ← AL0 J;*              SET RESULT 1
```

\*

\*\*\*\*\*

\*12: NAND \*

\*\*\*\*\*

\*

```
NAND:         D2 ← RETURN;*              NEGATE IS MONADIC (NOT)
                CALL BAND;
                GOTO NEGATE;
```

\*

\*\*\*\*\*

\*13: NOR \*

\*\*\*\*\*

\*

```
NOR:          D2 ← RETURN;
                CALL BOR;
                GOTO NEGATE;
```

\*

\*\*\*\*\*

\*14: LESS \*

\*\*\*\*\*

\*

```
LESS:         MONITOR ← RETURN;
                CALL FLSUB;
                LINK ← MONITOR;
                GOTO SET1 IF AR0$B=1;*      B = A < 0
                GOTO SET0;
```

\*

\*\*\*\*\*

\*15: <= \*

\*\*\*\*\*

\*

```
LEQ:          MONITOR ← RETURN;
                CALL FLSUB;
                LINK ← MONITOR;
                GOTO SET1 IF AR0$B=1;
                GOTO LEQ1;
```

\*

\*

\*\*\*\*\*

```

*16: >=          *
*****
GEQ:             LINK ← LEQ;
                  GOTO SWAPREGISTERS;
*
*****
*17: GREATER     *
*****
*
GREATER:         LINK ← LESS;
                  GOTO SWAPREGISTERS;
*
*
*****
*20: EQUAL      :
*****
EQUAL:           MONITOR ← RETURN;
                  GOTO EQCHAR IF CHARF = 1;*CHARACTERS ?
                  CALL FLSUB;
LEQ1:            LINK ← MONITOR;
                  GOTO SET1 IF AR0$R = 0;
                  GOTO SET0 IF AR0$1=0;*
                  AR0 ← AR0 OR BIT1;*
                  AR0 R8;*
                  AR0 ← AR0 -22;*AR0-23 P1
                  LINK ← MONITOR;
                  GOTO SET1 IF AR0$0 = 0;
                  GOTO SET0;
*
EQCHAR:          LINK ← MONITOR;
                  AR0 ← AR0 EOR AL0;
                  GOTO SET1 IF AR0$0=0;
                  GOTO SET0;
*
*****
*SET1*
*****
SET1:            AR0 ← FLOATINGONE ;
                  AR1 ← ZERO J;
SET0:            AR0 ← 0 ;
                  AR1 ← ZERO J;
*****
*21: NOTEQUAL   *
*****
*
NOTEQUAL:        D2 ← RETURN;
                  CALL EQUAL;
                  GOTO NEGATE;
*
*
INDEXOF:         GOTO NOTYET;
*
```

```

*VERIFY THAT AR IS ARRAY.
RESHAPE:      C1 ← AL0 EOR VECTORCODE;
              GOTO ERROR IF C1 ≠ 0; *LEFT ARG MUST BE NUM VECTOR
              C4 ← AL1;
              PTR1 ← AR1; *AS USUAL
*COMPUTE LENGTH OF RESULT AND GIVE IT A HOME :
              VMA ← AL1 + 2;
              READIT;
              C1 ← VMD; *          COUNTER = #ELEMENTS
              VMA ← AL1 + 1;
              READIT;
              VMA ← AL1 + VMD; *    POINT BELOW LEFT ARRAY
              AL0 ← FLOATINGONE;
              AL1 ← 0;
              VMA ← VMA + 1; *    CAVE CANE LOADAR
NBR:          VMA ← VMA - 3; *    NEXT ELEMENT
              CALL LOADAR;
              CALL FMUL;
              AL0 ← AR0; *          SAVE PARTIAL RESULT
              AL1 ← AR1;
              C1 ← C1 - 1; *    DECREMENT COUNTER
              GOTO NBR IF C1 ≠ 0;
*THE # ELEMENTS OF RESHAPED ARRAY IS IN AL AND AR.
              CALL FIX;
              GOTO ERROR IF INTOVERF = 1;
              C3 ← AR1; *          SAVE
*NOW GET THE HOME FOR THE RESULT:
              VMA ← PTR1;
              READIT;
              GOTO CHARSIZE IF VMD$4 = 1; *RIGHT ARRAY ALPHA ?
*RA SCALAR: SIZE IS 2 AR + HEADERSIZE + DIM#
              AR1 ← AR1 L1; *          2 AR
              AR1 ← AR1 + HEADERSIZE;
              VMA ← C4 + 2;
              READIT;
              FREE ← VMD; *    GET # DIM = LA'S # ELEMNTS
              C1 ← AR1 + FREE; *    C1 IS ALL SET
              CALL GETBLOCK;
*VMD NOW POINTS BELOW NEW BLOCK AND C2 TO WORD 0
              PTR2 ← VMD;
              D4 ← C2;
*CREATE AND ENTER THE NEW HEADER:
*FREE MUST STILL BE INTACT HERE.
*TURNON CHAR BIT IF APPLICABLE
              VMA ← C2;
              READIT;
              VMD ← VMD OR FLOATINGB;
              VMD ← VMD + FREE;
              WRITEIT;
              VMA ← C2 + 2;
              VMD ← C3; *          # ELEMNTS
              WRITEIT;
*NOW ENTER # ELEMENTS PER DIMENSION :

```

```

*DIMENSIONS ARE OBTAINED BY FIXING THE ELEMENTS OF THE LA.
*FREE CONTAINS THE # DIM = COUNTER.
*C2 IS THE RUNNING POINTER. SET IT:
    C1 ← C2 + HEADERSIZE;* INITIAL DEPOSIT ADDRESS
    VMA ← C4 + 1;
    READIT;
    C2 ← C4 + VMD;
    D1 ← FREE;
DIMIN:    C2 ← C2 - 2;*          NEXT
    VMA ← C2;
    CALL LOADAR;
    GOTO ERROR IF AR0$8 = 1;*MUST BE >0
    CALL FIX;
    VMA ← C1;
    VMD ← AR1;
    WRITEIT;
    C1 ← C1 + 1;
    D1 ← D1 - 1;
    GOTO DIMIN IF D1 ≠ 0;
*DIMENSIONS ARE IN.
*RELEASE NOW THE LA WHICH IS USELESS :
    RETURN ← RNBR;
    VMA ← C4;
    C1 ← RNBR;
    C2 ← RELEASEBLOCK;
    GOTO RELEASETST;
*ENTER THE ELEMENTS.
*3 CASES: FINAL ELEM# IS (<,<=,>,>) RA'S ELEM#.
*FINAL ELEM# IS IN C3.
*GET RA'S ELEM#:
* C3 CONTAINS RESULT'S # ELEMENTS
* D4 POINTS TO RESULT
*PTR2 POINTS UNDER RESULT
*C4 WILL BE # ELEMENTS OF RIGHT ARRAY
*D3 WILL BE RA TOP POINTER
RNBR:    VMA ← PTR1 + 2;
    READIT;
    C4 ← VMD;
*IF FINAL # IS <= JUST SET LENGTH(TRANSFER) :
    D3 ← PTR1;
    VMA ← PTR1 + 1;
    GO READ;
    PTR1 ← PTR1 + VMD;
    D1 ← PTR1;*          POINT BELOW RA
LENGTHTEST: FREE ← C3 EOR -1;*          -FINAL#
    VMD ← C4 + FREE P1;*          RA'S - FINAL = C4 - C3
    GOTO ONEXFER IF VMD$0 = 0;* OK FOR XFER IF C4 >= C3
* C3 > C4 MUST ITERATE.
*> : THE RA MUST BE EXTRAPOLATED.
    C1 ← C4 L1;*          COUNTER
    SPARED ← C1 EOR -1;
    PTR2 ← PTR2 + SPARED P1;
    PTR1 ← D1 + SPARED P1;* MOVE BOTH POINTERS UP
    D2 ← PTR2;

```

```

        CALL XFERLOOP;
        PTR2 ← D2;*
        SPARED ← C4 EOR -1;
        C3 ← C3 + SPARED P1;*
        GOTO LENGTHTEST;
        NEW BASE FOR RESULT
        C3 - C4
*
CHARSIZE:    GOTO NOTYET;
*
*   C3 ≤ C4
ONEXFER:     C1 ← C3 L1;
             SPARED ← C1 EOR -1;
             PTR1 ← D1 + SPARED P1;
             PTR2 ← PTR2 + SPARED P1;
             CALL XFERLOOP;
             SETSTATE1;
             AR1 ← D4;*
             VMA ← D4;
             GO READ;
             VMD ← VMD AND KILL4;*
             AR0 ← VMD OR APCODE;
             VMA ← D3;*
             GOTO RELVEC2;
             POINTER TO RESULT
             EXTRACT # DIMENSIONS
             POINTER TO RA FOR RELEASE
*
*
RAVEL:       GOTO NOTYET;
COMPRESS:    GOTO NOTYET;
EXPAND:      GOTO NOTYET;
EXPO:        GOTO NOTYET;
XPAND:       GOTO NOTYET;
SLASH:       GOTO NOTYET;
XSLASH:      GOTO NOTYET;
SIZE:        GOTO NOTYET;
CATENATE:    GOTO NOTYET;
POWER:       GOTO NOTYET;
DOTOPER:     GOTO NOTYET;
*
*
```

```

*****
*
*
*****      M O N A D I C      O P E R A T O R S      *****
*
** C4 IS UNAVAILABLE : SAVED NEXT OPERATOR.
*
*****
*01:      -      *
*****
*
UMINUS:      GO BRKUPA;
              GO COMPA;
              LINK ← RETURN;
              GOTO NPACK;

*
*****
*      $TIM      *
*****
*
SIGNUM:      LINK ← RETURN;
              GOTO SET1 IF AR0$8 = 0;* ANSWER IS 0 IF >0
SETM1:      AR0 ← 200B;
              AR1 ← ZERO J;

**
*****
*      $DIV      *
*****
*
RECIPROCAL:  AL0 ← FLOATINGONE;
              AL1 ← 0;
              GOTO FDIV;

*
*****
*      $STOP      *
*****
*
CEILING:      D3 ← RETURN;
              GOTO NEGCEILING IF AR0$8 = 1;* DIFFERENT CASE IF <0
POSFLOOR:    CALL VERIFYINTEGER;
              LINK ← D3;
              GOTO MUSTADDONE IF AR0 = 0;* WAS NOT INTEGER
              AR0 ← 0;*      WAS INTEGER
              AR1 ← D4 ;
              RETURN ← LINK;
              GOTO FLOAT;*RECONSTITUTED INTEGER JUICE
MUSTADDONE:  AR1 ← D4 + 1;*  ADD ONE TO FIXED INTEGER
              AR0 ← 0;*  RESTORE THE INETEGEER AND ADD ONE
              RETURN ← D3;
              GOTO FLJAT;

*
NEGCEILING:  D3 ← RETURN;
              GO FUZZAR1;

```



```

        CALL FIX;
        RETURN ← D3;
        GOTO FLOAT;

*
*****
*   $BOT   *
*****
*
FLOOR:      GOTO NEGCEILING IF AR0$8 = 0;
            GOTO POSFLOOR;

*
*****
*   $BAR   *
*****
*
MAGNITUDE:  LINK ← RETURN;
            GOTO UMINUS IF AR0$8 = 1;
            GOTO OUT;

*
*****
*   $NOT   *
*****
*
NEGATE:     RETURN ← D2;
NOT:        LINK ← RETURN;
            GOTO SET1 IF AR0$R = 0; * IS IT A 0 ?
            GOTO SET0;

*
**
*****
*   IOTA   *
*****
*
IOTA:       TURNON SCASCAB; * NO VECTOR OPERAND
            GOTO ERROR IF AR0$0#0; * REQUIRES INTEGER ARG
            GOTO ERROR IF AR0$8 = 1; * >0 ARG
            GOTO NULL IF AR0$R = 0; * IOTA 0 IS NULL
            CALL VERIFYINTEGER;
            GOTO ERROR IF AR0=0; * WAS IT AN INTEGER?
            C1 ← 0;
            C2 ← D4;

*INTEGR IS IN C1 C2 . C2 IS USED AS A COUNTER
            C3 ← 0; * INITIALIZE FOR LOOP = CURRENT INTEGER
*THE BLOCK TO BE CREATED IS OF DIMENSION C2 * 2 + 5
*GET A BLOCK OF THE APPROPRIATE SIZE-
*SET PTR1 BELOW IT
*C1 MUST CONTAIN THE # WORDS TO BE ALLOCATED
*GETBLOCK RETURNS THRU RETURN WITH
*VMD POINTING BELOW NU BLOCK
* C2 TO WORD 0.
            FREE ← C2; *           IOTA OF FREE
            C1 ← C2 L1; *          TRIVIAL
            C1 ← C1 + 5; *          SIZE OF NU BLOCK
            CALL GETBLOCK;

```

```

PTR1 ← VMD;*                                BELOW BLOCK
*CREATE HEADER FOR BLOCK:
VMA ← C2;*POINTER TO WORD 0
*CREATE WORD 0.
*IN FINAL ,PRESERVE P BIT IN WORD 0.
VMD ← 40001B;*                                # DIM
GO WRITE;
VMA ← VMA + 1;
VMD ← C1;*                                SIZE
GO WRITE;
VMA ← VMA + 1;
VMD ← FREE;*                                # ELEMENTS
GO WRITE;
VMA ← VMA + 1;
VMD ← ZERO;*                                DEFERMENT FIELD
GO WRITE;
VMA ← VMA + 1;
VMD ← FREE;*                                #ELEM FOR DIM 1
GO WRITE;
C1 ← FREE;* SAVE IT SINCE FLOAT DESTROYS FREE
*
*HEADER IS OK . GENERATE VECTOR
IOTALOOP: C3 ← C3 + 1;
AR1 ← C3;*                                NEXT INTEGER
AR0 ← 0;
CALL FLOAT;
PTR1 ← PTR1 - 2;
VMA ← PTR1;
VMD ← AR0;
GO WRITE;
VMA ← VMA + 1;
VMD ← AR1;
GO WRITE;*ENTER IT.
FREE ← C1;
SPARED ← C3 EOR FREE;
GOTO IOTALOOP IF SPARED # 0;
AR0 ← 100001B;
AR1 ← C2;
GOTO OPRETURN;
*
*
```

\*ROUTINES:

\*

VERIFYINTEGER: MONITOR ← RETURN;

AL0 ← AR0;

AL1 ← AR1;\* SAVE THE ORIGINAL

CALL FIX;

D4 ← AR1;\* THE INTEGER, CONCENTRATED FROM THE REAL THING

CALL FLOAT;\* RECONSTITUTE NUMBER

RETURN ← MONITOR;

GOTO EQUAL;\* COMPARE.

\*

\*

\*

\*

FUZZARI:

GOTO DOWNFUZZ IF AR1\$14 = 0;\* 1- ?

GOTO 0 IF AR1\$15 = 0 IX;\* DO NOTHING IF 10

SPARED ← AR1 AND KILL10;

SPARED ← SPARED EOR 77B;

GOTO 0 IF SPARED # 0 IX;

AL0 ← AR0 AND MASK8;\*

EXTRACT EXPONENT

AL0 ← AL0 R8;

AL0 ← AL0 - 23;\* ONE IN 23 TH BIT POSITION ????????????

AL0 L8;\* RESTORE AS EXPONENT

AL0 ← AL0 OR BIT9;\* ONE

AL1 ← 0;

RETURN ← LINK;

GOTO FLADD;\* ADD 1 IN LEAST SIGNIFICANT BIT POSITION.

\*

DOWNFUZZ:

AR1 ← AR1 AND FUZZM J\* 01 BECOMES 00.

FINISHED:

ZERO J;

\*

END;

COMPLETE TIMING FOR ARRAY ADDITION

```

GO SIMUL;
FIRSTLINE
DTE; 3
OPR;
STARTLINE
DTE; 5
OPR;
PUSH
DTE; 8
OPR;
CLEAR
DTE; 41
OPR;
PARSENEXT
DTE; 42
OPR;
GETTOKEN
DTE; 43
OPR;
DECYPHER
DTE; 52
OPR;
ARRAY
DTE; 56
OPR;
O1YPECHEC
DTE; 70
OPR;
OUTIN1
DTE; 75
OPR;
PARSENEXT
DTE; 78
OPR;
GETTOKEN
DTE; 79
OPR;
DECYPHER
DTE; 88
OPR;
OPERATOR
DTE; 92
OPR;
OUTIN2
DTE; 98
OPR;
PARSENEXT
DTE; 101
OPR;
GETTOKEN
DTE; 102
OPR;
DECYPHER
DTE; 111
OPR;
ARRAY
DTE; 115
OPR;
O1YPECHEC
DTE; 129
OPR;
BINOPER
DTE; 131
OPR;

```

N2

```

OP1YPETES
DTE; 137
OPR;
XXXVEC
DTE; 144
OPR;
VECVEC
DTE; 151
OPR;
PUSH
DTE; 154
OPR;
DIMCHECK
DTE; 165
OPR;
INTESTB
DTE; 227
OPR;
NONRELEAS
DTE; 240
OPR.1;
NONRELEAS
DTE; 260
OPR;
GETBLOCK
DTE; 275
OPR;
SCALOOP
DTE; 405
ICE MADDR,2023,2030;
MADDR,3747: 0
MADDR,3750: 100001
MADDR,3751: 17
MADDR,3752: 5
MADDR,3753: 0
MADDR,3754: 5
MADDR,3755: 0
MADDR,3756: 0
OPR;
GETLSCALA
DTE; 407
OPR;
GETRSCALA
DTE; 432
OPR;
DYADIC
DTE; 458
OPR;
FLADD
DTE; 463
ROAL0,AL1,OPR,AR0,AR1;
AL0(30): 500
AL1(31): 0
OPR(34): 120000
AR0(24): 1100
AR1(25): 0
OPR;
OUT
DTE; 505
RO AR0,AR1;
AR0(24): 1140
AR1(25): 0

```

N3

QPR;  
OUT  
QTE; 533  
QPR;  
SCALOP  
QTE; 535  
QPR;  
GETLSCALA  
QTE; 537  
QPR;  
GETRSCALA  
QTE; 562  
QPR;  
DYADIC  
QTE; 588  
QPR;  
FLADD  
QTE; 593  
QPR;  
OUT  
QTE; 630  
QPR;  
OUT  
QTE; 658  
QPR;  
SCALOP  
QTE; 660  
QPR;  
GETLSCALA  
QTE; 662  
QPR;  
GETRSCALA  
QTE; 687  
QPR;  
DYADIC  
QTE; 713  
QPR;  
FLADD  
QTE; 718  
QPR;  
OUT  
QTE; 760  
QPR;  
OUT  
QTE; 788  
QPR;  
SCALOP  
QTE; 790  
QPR;  
GETLSCALA  
QTE; 792  
QPR;  
GETRSCALA  
QTE; 817  
QPR;  
DYADIC  
QTE; 843  
QPR;  
FLADD  
QTE; 848

```

OPR;
OUT
OTE; 885
OPR;
OUT
OTE; 913
OPR;
SCALOOP
OTE; 915
OPR;
GETLSCALA
OTE; 917
OPR;
GETRSCALA
OTE; 942
OPR;
DYADIC
OTE; 968
OPR;
FLADD
OTE; 973
OPR;
OUT
OTE; 1010
OPR;
OPRETURN
OTE; 1037

```

N4

```

CE MADDR,2024,2040;
MADDR,3750: 100001
MADDR,3751: 17
MADDR,3752: 5
MADDR,3753: 0
MADDR,3754: 5
MADDR,3755: 2130
MADDR,3756: 0
MADDR,3757: 2110
MADDR,3760: 0
MADDR,3761: 1560
MADDR,3762: 0
MADDR,3763: 1520
MADDR,3764: 0
MADDR,3765: 1140
MADDR,3766: 0
MADDR,3767: 0
MADDR,3770: 0
0

```



N5

```

    TE;    1037
    QPR;
    POP
    QTE;    1042
    QPR;
    RELEASETE
    QTE;    1074
    QPR;
    NONRELEASE
    QTE;    1087
    QPR;
    SECONDARR
    QTE;    1089
    QPR;
    RELEASETE
    QTE;    1096
    QPR;
    NONRELEASE
    QTE;    1105
    QPR;
    SECONDARR
    QTE;    1107
    QPR;
    PARSENEXT
    QCE 2024,2044;
    MADDR,3750:  140001
    MADDR,3751:  17
    MADDR,3752:  5
    MADDR,3753:  0
    MADDR,3754:  5
    MADDR,3755:  2130
    MADDR,3756:  0
    MADDR,3757:  2110
    MADDR,3760:  0
    MADDR,3761:  1560
    MADDR,3762:  0
    MADDR,3763:  1520
    MADDR,3764:  0
    MADDR,3765:  1140
    MADDR,3766:  0
    MADDR,3767:  0
    MADDR,3770:  0
    MADDR,3771:  0
    MADDR,3772:  0
    MADDR,3773:  0
    MADDR,3774:  0
    QRO AR0,AR1;
    AR0(24):  100001
    AR1(25):  3750
    QPE+RTE
    Q;

```

NOT A COMMAND

OTE; 1112

OPR;

GETTOKEN

OTE; 1113

OPR;

DECYPHER

OTE; 1122

OPR;

OPERATOR

OTE; 1126

OPR;

EXECOP

OTE; 1130

OPR;

LD

OTE; 1134

OPR;

POP

OTE; 1137

OPR;

PRINTAR

ORO AR0,AR1;

AR0(24): 100001

AR1(25): 3750

OPR;

NEXTLINE

N6

325

OCMADDR,2024,2040;

MADDR,3750: 140001

MADDR,3751: 17

MADDR,3752: 5

MADDR,3753: 0

MADDR,3754: 5

MADDR,3755: 2130

MADDR,3756: 0

MADDR,3757: 2110

MADDR,3760: 0

MADDR,3761: 1560

MADDR,3762: 0

MADDR,3763: 1520

MADDR,3764: 0

MADDR,3765: 1140

MADDR,3766: 0

MADDR,3767: 0

MADDR,3770: 0

0

# TIMING OF A RECURSIVE FUNCTION

## Appendix O Timing of a Recursive Function

TE is Time Elapsed in tenths of microseconds

TYPE '0' TO DELETE A LINE  
 TYPE 'END' WHEN FINISHED

MELA-APL TRANSLATOR READY...

OK	\$DEL Z ← FAC N
OK	[1] \$GOTO 4 \$TIM \$GOTA N = 0
OK	[2] Z ← N \$TIM FAC N - 1
OK	[3] \$GOTO 0
OK	[4] Z ← 1
OK	[5] \$DEL
OK	FAC 9
OK	FAC 1
OK	FAC 2
OK	FAC 3

END

EXECUTION REQUESTED?

NO  
INTERNAL APL STRINGS READY ON :TEST\*:~:

```

*$DEL Z ← FAC N
* RESULT1 (R)      = 1
* ARG# (AA)        = 1
* PARAMETER #      = 0
* LOCALPHANTOMS #  = 2
*-----*
* NEW FUNCTION : FAC !      FUNCTION # = 5
*-----*
*$GOTO 4 $IIM $IOTA N = 0
      CORE MADDR,650:000000B,000000B,120220B,110000B,121023B,
      120002B,001500B,000000B,122000B,122004B;
*Z ← N $IIM FAC N - 1
      CORE MADDR,660:000500B,000000B,120001B,110000B,114000B,
      120002B,110000B,122005B,110001B,122004B;
*$GOTO 0
      CORE MADDR,670:000000B,000000B,122000B,122004B;
*Z ← 1
      CORE MADDR,674:000500B,000000B,122005B,110001B,122004B;
*$DEL
*LOCAL VARIABLES NUMBER IS 2
*
*****
*THE LOCAL PHANTOM TABLE:
* 0      = 110000 = N
* 1      = 110001 = Z
*****
*
*LABELS NUMBER IS : 0
*
*LOAD GFRZ WITH A NEW FUNCTION CALL TO FAC :
      CORE MADDR,4089: 160006B,120000B;
*ENTRY IN FUNCTION TABLE:
      CORE MADDR,224: 120000B,2B,1025B,0B;
*THE LINE TABLE IS :
      CORE MADDR,532: 4,0,650,660,670,674;
*
*FAC 9
      CORE MADDR,679:002110B,000000B,114000B,122004B;
*FAC 1
      CORE MADDR,683:000500B,000000B,114000B,122004B;
*FAC 2
      CORE MADDR,687:001100B,000000B,114000B,122004B;
*FAC 3
      CORE MADDR,691:001140B,000000B,114000B,122004B;
*****
*THE GLOBAL PHANTOM TABLE :
* 0      = 114000 = FAC
*****

```

Program

Strings

END

\$

END AT FINISHED

0SBFONCALL,ERROR,FRETURN,CALCMODE;

0SBPRINTAR,STARTLINE,PARSENEXT,PHANTOM,GETPHANTOM,SCALAR;

0SBPHIPROCESSOR,PARAMCHECK,NOPARAM;

0SBERASELOOP,NOVECTOR,NUFRAME,LFRZCLEAN,ENTERUV,LD,NUHISTORY;

0SBENTERLABELS;

0RFPP,679;

0GO TEST

329

Breakpoints

;

CALCMODE

0IE; 4

0PR;

PARSENEXT

0IE; 13

0PR;

SCALAR

0IE; 29

0PR;

PARSENEXT

0IE; 50

0PR;

PHANTOM

0IE; 65

0PR;

GETPHANTO

0IE; 67

0PR;

PHIPROCES

0IE; 94

0PR;

FONCALL \_\_\_\_\_ FAC 9

0E;

NOT A COMMAND

0IE; 98

0PR;

PARAMCHECK

0IE; 103

0PR;

NOPARAM

0IE; 104

0PR;

NUFRAME

0IE; 148

0PR;

ENTERUV

0IE; 198

0PR;

ENTERUV

0IE; 204

0PR;

NUHISTORY

0IE; 205

0PR;

ENTERLABEL

0IE; 311

0

check syntax and  
set up new  
LFRZ frame.

```

    PR;
STARILINE
    DIE;    330
    QSB IOTA,FMUL,BRANCH;
    DPR;
PARSENEXT
    DIE;    367
    DPR;
SCALAR
    DIE;    378
    DPR;
PARSENEXT
    DIE;    399
    DPR;
PARSENEXT
    DIE;    423
    DPR;
PHANTOM
    DIE;    438
    DPR;
GETPHANTO
    DIE;    440
    DPR;
PHIPROCES
    DIE;    468
    DPR;
PARSENEXT
    DIE;    546
    DPR;
PARSENEXT
    DIE;    570
    DPR;
IOTA
    DIE;    606
    DPR;
PARSENEXT
    DIE;    631
    DPR;
SCALAR
    DIE;    642
    DPR;
PARSENEXT
    DIE;    681
    DPR;
BRANCH
    DIE;    704
    DPR;
STARILINE
    DIE;    750
    0

```

```

CB;                                clear breakpoints
[SBFCNCALL,FRETURN,STARTLINE,CALCMODE,PRINTAR,LFNZCLEAN,ERASELOPP,NOVEC]
QPR;
FCNCALL                            FAC  8
QIE;  1026
QPR;
STARTLINE
QIE;  1258
QPR;
NOT A COMMAND
QPR;
STARTLINE
QIE;  1678
QPR;
FCNCALL                            FAC  7
QIE;  1960
QPR;
STARTLINE
QIE;  2192
QPR;
STARTLINE
QIE;  2612
QPR;
FCNCALL                            FAC  6
QIE;  2885
QPR;
STARTLINE
QIE;  3117
QPR;
STARTLINE
QIE;  3537
QPR;
FCNCALL                            FAC  5
QIE;  3810
QPR;
STARTLINE
QIE;  4042
QPR;
STARTLINE
QPR;
FCNCALL                            FAC  4
QIE;  4735
QPR;
STARTLINE
QIE;  4967
QPR;
STARTLINE
QIE;  5387
QPR;
FCNCALL                            FAC  3
QIE;  5666
QPR;
STARTLINE
QIE;  5898
QPR;
STARTLINE
QIE;  6318
Q

```



```

PR;
FCNCALL
01E; 6588
QPR;
STARTLINE
01E; 6820
QPR;
STARTLINE
01E; 7240
QPR;
FCNCALL
01E; 7516
QPR;
STARTLINE
01E; 7748
QPR;
STARTLINE
01E; 8168
QPR;
FCNCALL
01E; 8419
QPR;
STARTLINE
01E; 8651
QPR;
STARTLINE
01E; 10124
QPR;
FRETURN
01E; 10342
QPR;
ERASELOOP
01E; 10343
QPR;
NOVECTOR
01E; 10357
QPR;
ERASELOOP
01E; 10359
QPR;
NOVECTOR
01E; 10373
QPR;
ERASELOOP
01E; 10375
QPR;
LFRZCLEAN
01E; 10378
QPR;
STARTLINE
01E; 10888
QCB ERASELOOP,NOVECTOR;
QPR;
FRETURN
01E; 11073
QPR;
LFRZCLEAN
01E; 11109
QPR;
STARTLINE
01E; 11619
Q

```

FAC 2

FAC 1

FAC 0

1

1

192.3 $\mu$ s355.7 $\mu$ s

PR;	2
FRETURN	
QIE; 11804	
QPR;	
LFRZCLEAN	
QIE; 11846	
QPR;	
STARTLINE	
QIE; 12350	
QPR;	6
FRETURN	
QIE; 12535	
QPR;	
LFRZCLEAN	
QIE; 12571	
QPR;	
STARTLINE	
QIE; 13081	
QPR;	24
FRETURN	
QIE; 13266	
QPR;	
LFRZCLEAN	
QIE; 13302	
QPR;	
STARTLINE	
QIE; 13812	
QPR;	120
FRETURN	
QIE; 13997	
QPR;	
LFRZCLEAN	
QIE; 14033	
QPR;	
STARTLINE	
QIE; 14537	
QPR;	720
FRETURN	
QIE; 14722	
QPR;	
LFRZCLEAN	
QIE; 14758	
QPR;	
STARTLINE	
QIE; 15261	
QPR;	5040
FRETURN	
QIE; 15446	
QPR;	
LFRZCLEAN	
QIE; 15482	
QPR;	
STARTLINE	
QIE; 15991	
QPR;	40320
FRETURN	
QIE; 16176	
QPR;	
LFRZCLEAN	
QIE; 16212	
QPR;	
STARTLINE	0 - 7
QIE; 16721	
Q	

```

PR;
FRETURN
DTE; 16906
DPR;
LFRZCLEAN
DTE; 16942
DPR;
PRINTAR
DTE; 17064 ←
DRO AR0,AR1;

```

```

AR0(24): 11530
AR1(25): 114000

```

362880

Timing for FAC 1

```

;
CALCMODE
DIF;
DTE; 0 ←
DPR;
FCNCALL
DTE; 94
DPR;
FCNCALL
DTE; 997
DPR;
FRETURN
DTE; 2920
DPR;
FRETURN
DTE; 3651
DPR;
PRINTAR
DTE; 3809 ←
DRO AR0,AR1;

```

```

AR0(24): 500
AR1(25): 0

```

192.3 $\mu$ s355.7 $\mu$ s

1

CALC MODE

01F3

01E3 0

0PR3

FCNCALL

01E3 94

0PR3

FCNCALL

01E3 1022

0PR3

FCNCALL

01E3 1925

0PR3

FRETURN

01E3 3848

0PR3

FRETURN

01E3 4579

0PR3

FRETURN

01E3 5310

0PR3

PRINIAR

01E3 5468

0RD AR0,AR1;

AR0(24): 1100

AR1(25): 0

0

2

FAC 3

PR3

CALC MODE

01F3

01E3 0

0PR3

FCNCALL

01E3 94

0PR3

FCNCALL

01E3 1016

0PR3

FCNCALL

01E3 1944

0PR3

FCNCALL

01E3 2847

0PR3

FRETURN

01E3 4770

0PR3

FRETURN

01E3 5501

0PR3

FRETURN

01E3 6232

0PR3

FRETURN

01E3 6963

0PR3

PRINIAR

01E3 7121

0RD AR0,AR1;

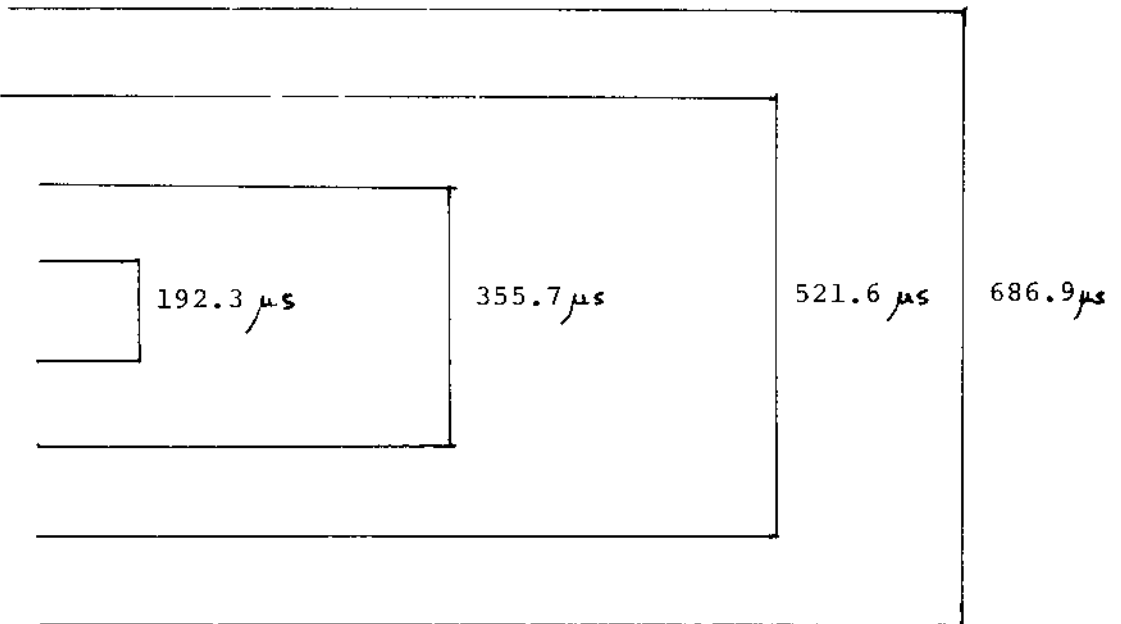
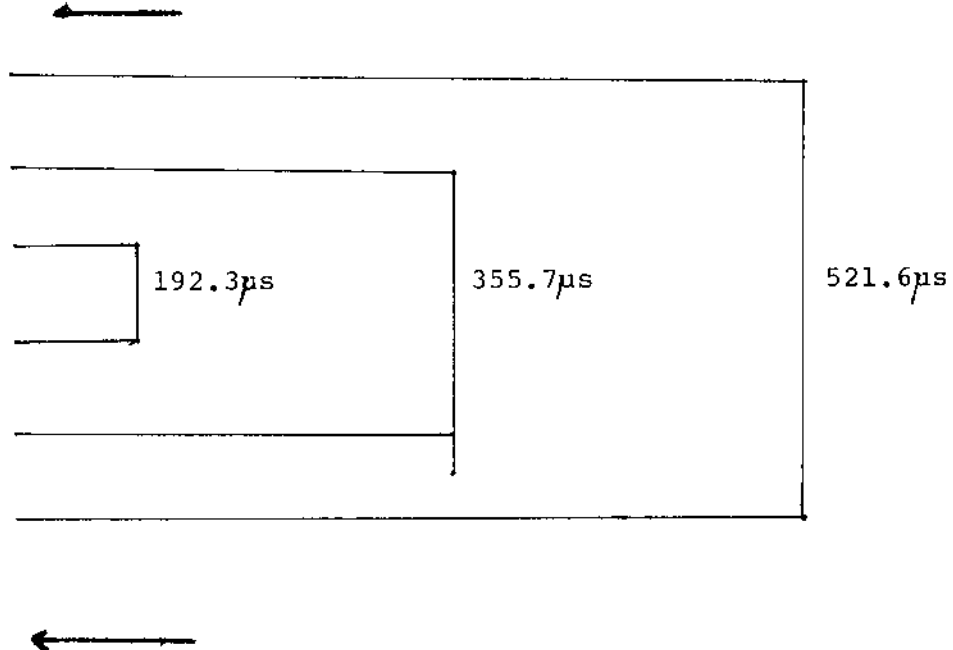
AR0(24): 1540

AR1(25): 0

0

6

0 - 9



## **BIBLIOGRAPHY**

## BIBLIOGRAPHY

- [1] Abrams, P. [1970], "An APL Machine," SLAC Report No. 114,  
Stanford University, February 1970.
- [2] Anderson, J.P. [1961], *A Computer for Direct Execution of  
Algorithmic Languages*, Eastern JCC, Macmillan Company,  
New York, 1961, pp. 184-193.
- [3] Bashkow, T.R., Sasson, A., and Kronfeld, A. [1967], "System  
Design of a FORTRAN Machine," *IEEE Transactions on  
Electronic Computers*, Vol. EC-16, No. 4, August 1967,  
pp. 485-499.
- [4] Berges, G.A. and Rust, F. [1969], *APL/MSU Reference Manual*,  
Department of Electrical Engineering, Montana State  
University, Bozeman, Mont., April 1969.
- [5] Berry, P. [1969], *APL/360 Primer*, Form No. C20-17202-0, IBM,  
White Plains, New York, 1969.
- [6] Breed, L.M., Lathewell, R.H. [1968], *The Implementation of  
APL/360, Interactive Systems for Applied Mathematics*,  
Academic Press, New York, 1968, pp. 390-399.
- [7] Carberry, R.S. et al. [1968], *A Programming Language/1130*,  
IBM Contributed Library 1130 03.3.001, 1968.
- [8] Cook, R.W., Flynn, M.J. [1970], "System Design of a Dynamic  
Microprocessor," *IEEE Trans.* C-19, 3, March 1970,  
pp. 213-222.
- [9] DeRemer, F.L. [1971], "Simple LR(k) Grammars," *Comm. ACM*, 14,  
7, July 1971, pp. 453-460.
- [10] Earley, J. [1969], "A Practical Incremental LR(1) Parsing  
Algorithm", Department of Computer Science, University  
of California, Berkeley, 1969.

- [11] Falkoff, A.D., Iverson, K.E. [1968], *APL/360 Terminal System, Interactive Systems for Applied Mathematics*, Academic Press, New York, 1968, pp. 22-37.
- [12] \_\_\_\_\_ [1968], *APL/360: User's Manual*, IBM, Yorktown Heights, New York, July 1968.
- [13] Feldman, J., Gries, D. [1968], "Translator Writing Systems," *CACM*, 11, 2, February 1968, pp. 77-113.
- [14] Floyd, R.W. [1963], "Syntactic Analysis and Operator Precedence," *JACM*, Vol. 10, No. 3, July 1963, pp. 316-333.
- [15] Flynn, M.H., MacLaren, M.D., "Microprogramming Revisited," *Proc. ACM*, 22nd National Conference, Psychonetics, Narbeth, Pennsylvania, pp. 457-464.
- [16] Foster, G.H. [1969], "APL: a perspicuous language," *Computers and Automation*, November 1969, pp. 24-28.
- [17] Green, J. [1966], "Microprogramming, Emulators and Programming Languages," *CACM* 9, 3, March 1966, pp. 230-231.
- [18] Hassitt, A., Lageschulte, J. and Lyon, L. [1971], "Implementation of a high level language machine," Fourth Workshop on Microprogramming, Santa Cruz, September 1971.
- [19] Hassitt, A. and Lyon, L.E. [1972], "Efficient Evaluation of Array Subscripts of Arrays," *IBM Journal of Research and Development*, January 1972, pp. 45-57.
- [20] Ichbiah, J.D., Morse, S.P. [1970], "A Technique for Generating Almost Optimal Floyd-Evans Productions for Precedence Grammars," *Comm. ACM* 13, 8, August 1970, pp. 501-508.
- [21] Irons, E.T. [1969], "Structural Connections in Formal Languages," *Comm. ACM* 7, 2, February 1969, pp. 67-72.
- [22] Iverson, K.E. [1962], *A Programming Language*, John Wiley and Sons, New York, 1962.

- [23] Knuth, D.E. [1965], "On the Translation of Languages from Left to Right," *Inf. Contr.*, 8, October 1965, pp. 607-639.
- [24] Korenjak, A.J. [1969], "A Practical Method for Constructing LR(k) Processors," *Comm. ACM* 12, 11, November 1969, pp. 613-623.
- [25] Lawson, H.W. [1968], "Programming-Language-Oriented Instruction Streams," *IEEE Trans.* C-17, 5, May 1968, pp. 476-485.
- [26] Loeckx, J. [1970], "An Algorithm for the Construction of Bounded-Context Parsers," *Comm. ACM* 13, 5, May 1970, pp. 297-307.
- [27] McKeeman, W.M. [1967], *Language Directed Computer Design*, FJCC 67, Thompson Books, Washington, D.C., pp. 413-417.
- [28] Melbourne, A.J., Pugmire, J.M. [1965], "A Small Computer for the Direct Processing of FORTRAN Statements," *Computer Journal*, Vol. 8, April 1965, pp. 24-28.
- [29] Pakin, S. [1968], *APL 360 Reference Manual*, Science Research Associates, Inc., Chicago, Illinois, 1968.
- [30] Sugimoto, M. [1969], "PL/I Reducer and Direct Processor," *Proceedings of the 24th National Conference*, ACM, New York, 1969.
- [31] Thurber, K.J. and Myrna, J.M. [1970], "System Design for a Cellular APL Computer," *IEEE Trans.* C-19, No. 4, pp. 291-300.
- [32] Tucker, A.B. and Flynn, M.J. [1971], "Dynamic Microprogramming: Processor Organization and Programming," *CACM* 14, 4, April 1971, pp. 240-250.



- [33] Weber, M. [1967], "A Microprogrammed Implementation of EULER on IBM System/360, Model 30," *CACM* 10, No. 9, September 1967, pp. 549-558.
- [34] Wirth, N., Weber, H. [1966], "EULER: A Generalization of ALGOL and its Formal Definition," *CACM* 9, No. 1, January 1966, pp. 13-23; *CACM* 9, No. 2, February 1966, pp. 89-99; *CACM* 9, No. 12, December 1966, p. 878.
- [35] Zaks, H.R. [1971], "A Language Machine," Third APL Conference, Berkeley, California, April 1971.
- [36] \_\_\_\_\_ [1971], "Microprogrammed APL," *Fifth IEEE International Computer Society Conference Proceedings*, September 1971, p. 193.
- [37] \_\_\_\_\_ [1971], "Microprogrammed APL Implementation," Fourth Microprogramming Workshop, Santa Cruz.
- [38] Zaks, H.R., Steingart, D., and Moore, J. [1971], "Firmware APL Time-Sharing System," *SJCC 71, AFIPS Proceedings*, Vol. 38, AFIPS Press, May 1971, pp. 179-190.

## INDEX

INDEX**A**

addressing	46
allocation	85
AP: see array pointer	
APL	2
architecture	157
argument	101
array	77,85,167
_____ management	85
_____ pointer	76
_____ zone	80,88
assignment	25,31,86,92
autospecification	39

**B**

bibliography	336
blocks	77
braces	24,30,58,63,104
branch	24,30

**C**

calculator mode	42
command	57

**D**

deallocation	88
deferment	87
definitions	7
descriptors	47,76
dynamic structures	70

**E**

editing	42
execop (or executable operator)	103
expression level	104
extensions	74
external APL	42

**F**

factorial	66,108,168
fix-up pass	44
floating algorithm	90
_____ bit	93
_____ point	62,227
free space	80
FRZ: see Function Reference Zone	
function	70,101
_____ call	31,101
_____ reference zone	81,84
_____ storage	61
_____ table	59,61

**G**

global variable	70
_____ zone	80
grammar	7

**H**

history segment	101
HS: see History Segment	

**I**

identifier	45
indexing	122
input-output device	56

_____ register	254
interfunction communication	71
internal strings	17,42,68
interpreter	70,98,256
IOD: see Input-Output Device	
Iverson's Language	2

## L

label	46,59
language machine	154,159
LD: see Line Delimiter	
left paren	30,104
_____ brace	24,30,104
_____ bracket	24,30,104
line delimiter	25,105
_____ number	103
_____ table	60
_____ storage	61
listing of translator	192
_____ interpreter	256
_____ operators	214
LN: see Line Number	
LFRZ: see (Local) Reference Zone	
literal array	51,88
localbase	99
local variable	70
look-ahead	23

**M**

map	99,158,247,251
materialization	101
Meta 4	159,229
Meta APL	74
microprocessor	145,148
microcode	145
microprogrammed interpreter	145
microprogramming	3,145

**N**

naming	42
nested primitive language	8
NOOP	55
NPL: see Nested Primitive Language	
null	77
numeric operator	105

**O**

odometer	130
operand	75
operator	29,39,63,103
scalar operators	105,118
other operators	23,120
descriptor	53
front end	118
software operators	39,214

**P**

parameter	71
parser	

PPL	11
NPL	12
APL	20,29
performance	165
phantom	44,46,52
_____ materialization segment	81,83,99,101
_____ processor	101,185
PMS: see Phantom Materialization Segment	
pointers	81
PPL: see Primitive Language	
precedence	6
primitive language	8
processor	160
program strings	48,61,80

## R

registers	99,149,254
_____ management	27
result	94,103,107
return	114
_____ Reference Zone	75
right delimiter	24,104
ripple	131,133
row-major order	126

## S

saved stack pointer	93
scalar	49,63
semantics	39,100,118

side-effects	70
software operators	39,214
space management	85
stack	82
_____ descriptors	76
states	14
storage allocation	95
_____ deallocation	96
structures (static)	59,68
_____ (dynamic)	70,75
symbol table	42
synchronization	75
syntax	6
_____ descriptors	47
_____ of a PPL	9,177
_____ of a NPL	10
_____ of APL	18,180
system zone	80

## **T**

timing	43,320
Teletype representation	209
Translator	62,188,192

## **U**

undefined variable	101
--------------------	-----

UY: see Undefined Variable

## **V**

vector	51,62
virtual memory	80



# SYBEX PUBLICATIONS

## TEXTS

C200	Introduction to Personal and Business Computing
C201	Microprocessors, by Rodney Zaks
C202	Microprocessor Programming
C207	Microprocessor Interfacing Techniques, by Austin Lesea and Rodney Zaks
E8	Microprocessor Encyclopedia: 8-bits
E5	Microprocessor Encyclopedia: Bit-slice
IMD	International Microprocessor Dictionary (10 languages)
X1	Microprocessor Lexicon
Z10	Microprogrammed APL Implementation, by Rodney Zaks

## SELF-STUDY COURSES

Each course includes audio-cassettes plus a special book.

S1	Introduction to Microprocessors (2½ hours)
S2	Programming Microprocessors (2½ hours)
SB1	Microprocessors (13 hours)
SB3	Military Microprocessor Systems (6½ hours)
SB5	Bit-slice (7 hours)
SB6	Industrial Application Techniques (6 hours)
SB7	Microprocessor Interfacing Techniques (6½ hours)

## COURSES ON VIDEO-CASSETTES

V1	Microprocessors (13 hours)
V3	Military Microprocessor Systems (6½ hours)
V5	Bit-slice (7 hours)
V7	Microprocessor Interfacing Techniques (6½ hours)

## SEMINAR TRAINING BOOKS

B1	Microprocessors
B2	Microprocessor Programming and Microprogramming
B3	Military Microprocessor Systems
B5	Bit-slice
B6	Industrial Microprocessor Systems
B7	Microprocessor Interfacing Techniques
B9	Microprocessor Troubleshooting
S1	Introduction to Microprocessors
S2	Programming Microprocessors
S3	Designing a Microprocessor System
S4	Microprocessor Applications
S5	Equipment Selection and Evaluation
S6	Super Speed: Bit-slice





**\$25.00**



**EUROPE**

**313 rue Lecourbe, 75015 Paris, France  
Tel (1) 828 25 02 TLX 200858**

**USA**

**2020 Milvia, Berkeley, California 94704  
Tel (415) 848-8233 TLX 336311**